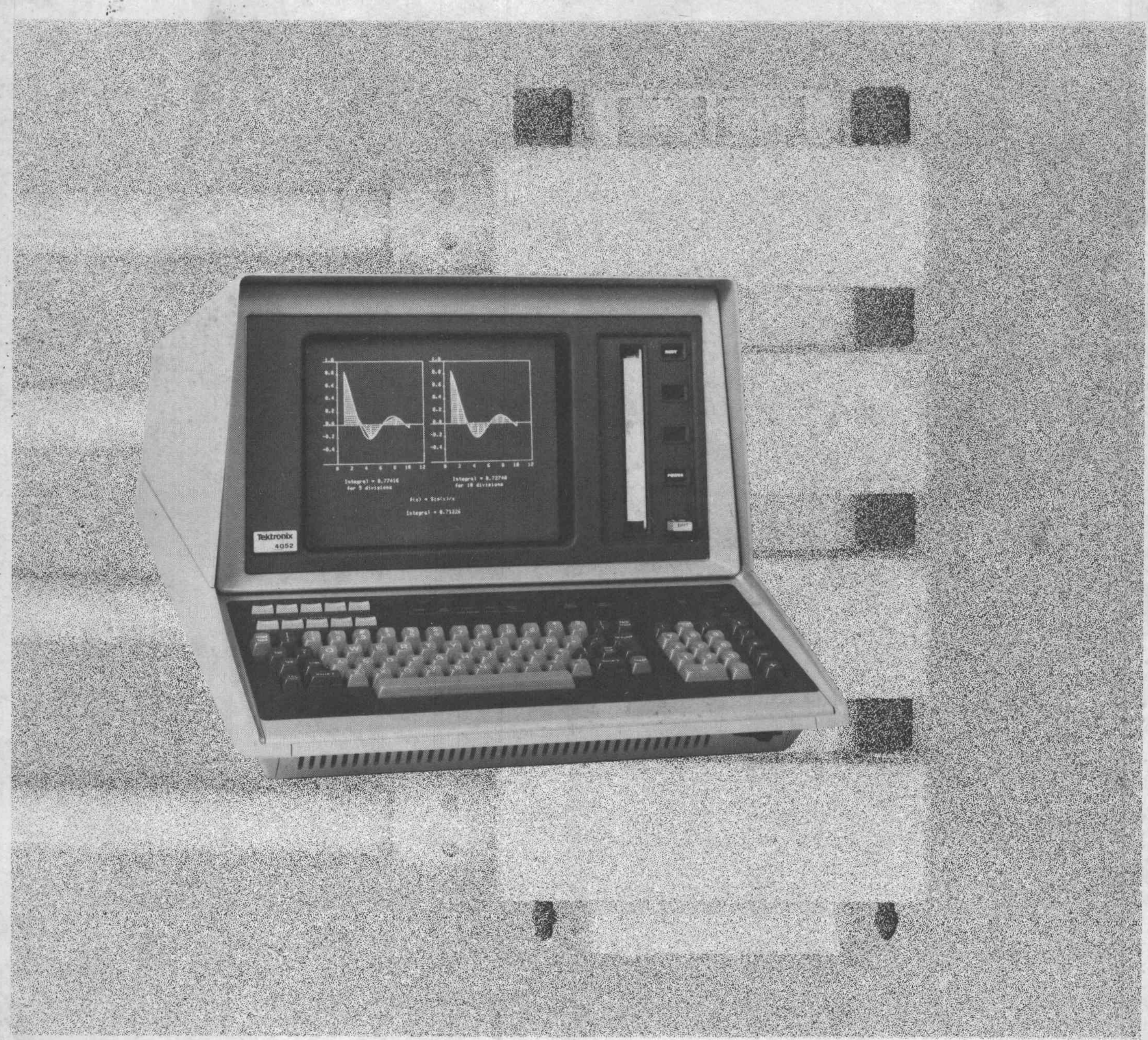
4052 GPIB Programming Guide



Tektronix®
COMMITTED TO EXCELLENCE

A binder is available from Tektronix for your GPIB Programming Guides. Contact your local Tektronix Field Office or representative and ask for part number 062-6433-00.

Additional Application and Programming Resources from Tektronix

- Application Engineers at many local field offices
- HANDSHAKE—Newsletter of Signal Processing and Instrument Control
- Applications Program Library
- Application Notes
- Other GPIB Programming Guides

For more information, contact your local Tektronix Field Office or representative.

This Programming Guide was written by Mark D. Tilden and produced by the Applications Support Group.

062-6400-00

First Printing JAN 1982

The GPIB can be a smooth path to automated test and measurement, or it can be a rough road, strewn with pitfalls. Choosing the right controller and instruments and writing efficient control programs can make the difference. This programming guide provides some guidelines for selecting system components and implementing a system based on the Tektronix 4052 Graphic Computing System.

Section 1 provides a brief introduction to the 4052's GPIB capabilities. Section 2 discusses guidelines for choosing system components and configuring the system. A brief review of the fundamentals of 4050 BASIC is provided in Section 3. Then, Section 4 gets down to the specifics of GPIB system programming with the 4052.

Section 5 is devoted to techniques for processing and displaying acquired data. Section 6 describes the factors that affect system performance and Section 7 provides some hints for improving the performance of the system.

Though this guide is based on the 4052 Graphic Computing System all of the information presented also applies to the 4054, and most applies to the 4051. Specific differences in the languages for the 4051, 4052 and 4054 are contained in the *4050 Series Graphic System Reference Manual.*

For complete information on the IEEE 488 bus (GPIB), refer to IEEE Standard 488-1978. Also, refer to the instrument manuals for specific information on programming GPIB instruments.

TABLE OF CONTENTS

TABLE OF CONTENTS

Section 1—The 4052 as a GPIB Controller

Defining the System Controller's Job

A typical GPIB system (Fig. 1-1) could include a controller, such as the Tektronix 4052 Graphic Computing System, a signal generator only able to listen, a digital counter, able to talk and listen, and a magnetic tape drive, only able to listen. These instruments can work together to perform a task, but they must be directed—and that's where the controller comes in.

At the heart of the GPIB system is its controller. In all but the simplest data logging applications, some form of controller is required to make the system work. But, taking full advantage of the controller's power requires a good understanding of it's job in the GPIB system. The controller's job can be broken into four major tasks:

1. Program development
2. Instrument control
3. Data processing
4. Display and storage

Program Development

The first task for many GPIB controllers is program development—writing, editing, and debugging the applications software that will control the system. This puts some special demands on the controller. For example, it should have a complete easy-to-use keyboard. In addition, user-definable keys, though not absolutely necessary, make menus or other input functions much simpler to implement and easier to use.

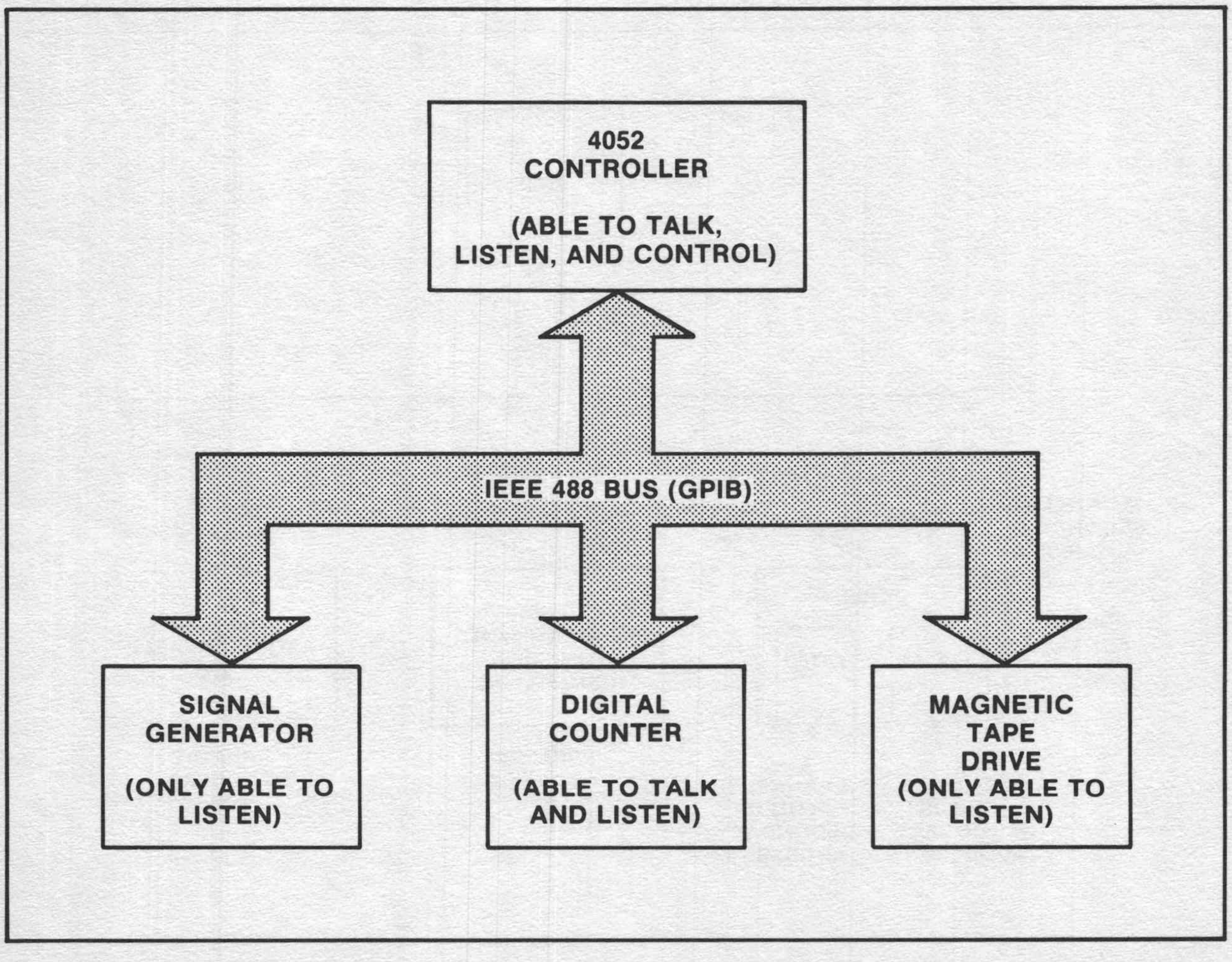


Fig. 1-1. *A typical GPIB system includes a controller and a variety of GPIB-interfaced devices with different capabilities.*

A full-size display is important for easy program listing and debugging. And, a convenient, powerful editor should be provided to create and modify program text. Finally, efficient mass storage is important for storing programs and data. A standard, transportable media allows programs developed on one controller to be transferred and run on many other systems.

Controlling the System

Next, consider the task of instrument control. No matter how powerful the system components are, if their actions are not coordinated, the system is like an orchestra without a conductor. The controller directs the entire system in performing its intended function. It assigns tasks to the instruments, coordinates communication, handles error conditions, and monitors the system's progress. The instrument control task can be further divided into four functions:

Addressing instruments
Sending commands
Transmitting and receiving data
Handling interrupts

Let's look at each of these functions individually:

Addressing instruments. The controller selects an instrument or set of instruments to be involved in an operation by addressing them. Every instrument is assigned a unique primary address in the range 0-30. The controller uses this address to assign a device to talk or listen. In addition, some instruments have secondary addresses that select sub-sections or functions within the instrument. For example, the Tektronix 7612D Programmable

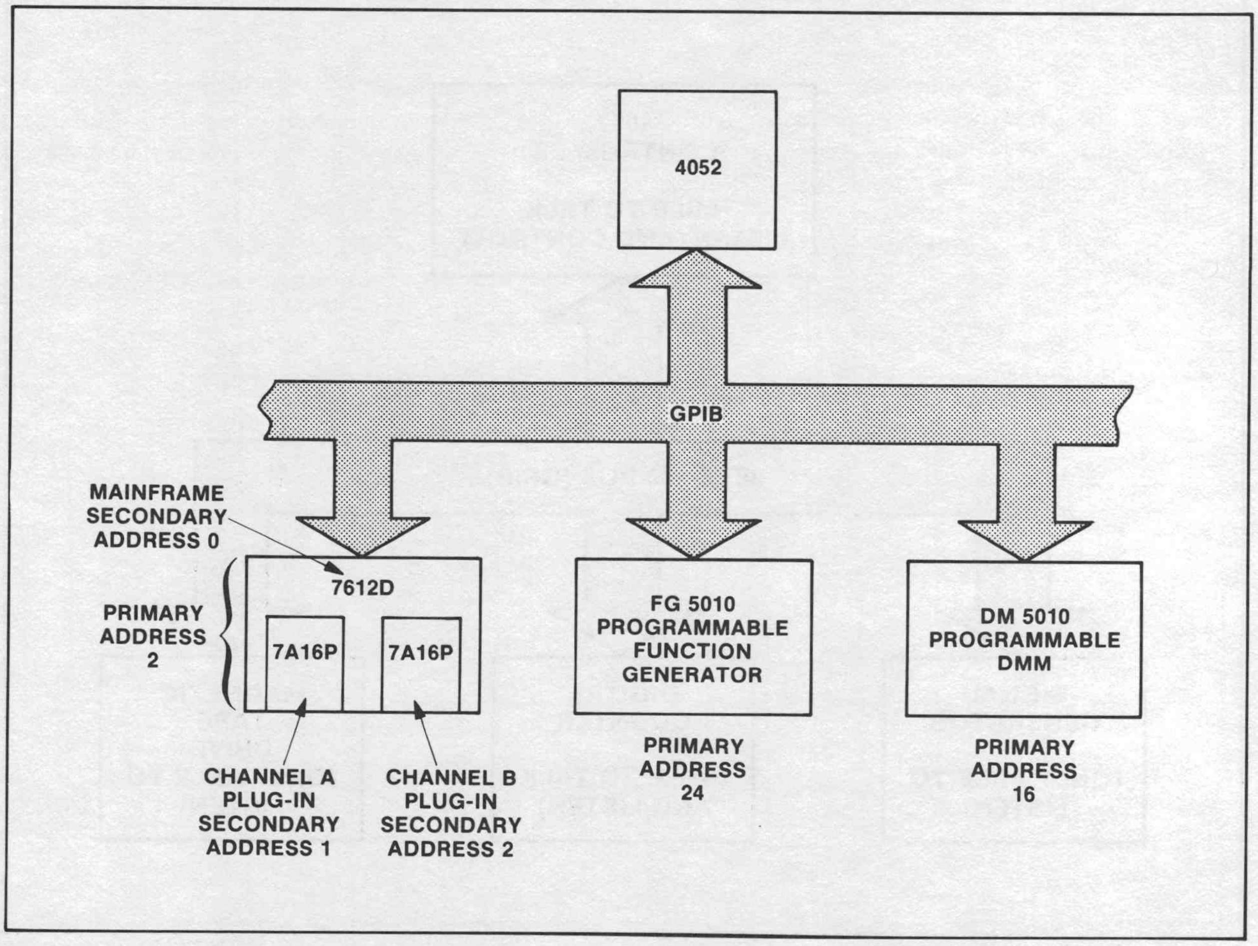

4052
GPIB
MAINFRAME SECONDARY ADDRESS 0
PRIMARY ADDRESS 2
7612D
7A16P
7A16P
CHANNEL A PLUG-IN SECONDARY ADDRESS 1
CHANNEL B PLUG-IN SECONDARY ADDRESS 2
FG 5010 PROGRAMMABLE FUNCTION GENERATOR
PRIMARY ADDRESS 24
DM 5010 PROGRAMMABLE DMM
PRIMARY ADDRESS 16

Fig. 1-2. *Each instrument on the bus is assigned a unique primary address. Secondary addresses are used in some instruments to select sub-sections or functions within the instrument.*

Digitizer has a secondary address for its mainframe and one for each programmable plug-in (Fig. 1-2).

Sending data and commands. The controller sends two basic types of messages: device-dependent messages and interface messages (Fig. 1-3). Device-dependent messages consist of commands or data that control instrument functions. The content and format of these messages is not specified in the IEEE 488 standard; it is left to the instrument designer. The messages may consist of queries that return instrument settings or data, commands that control instrument settings, or other data, such as waveforms. Device-dependent messages are always sent with the GPIB attention (ATN) line unasserted.

Interface messages are commands that control interface functions. They are sent only by the controller and always with ATN asserted. (Actually, the IEEE-488 standard defines the REN, ATN, and EOI bus lines as interface messages also. However, multi-line interface messages—messages sent by placing a byte on the data lines—can only be sent by the controller, and they are always sent with ATN asserted.) Interface messages may either be addressed commands, affecting only the addressed instruments; or universal commands, affecting all instruments on the bus.

Transmitting and receiving data. Most instruments send data to and/or receive data from the system controller. A digitizer, for example, acquires waveform data and transmits it to the controller for processing and storage. A programmable spectrum analyzer, such as the Tektronix 492P might receive processed waveform data from the controller for display on its CRT. Data may be transmitted using a variety of codes including binary or ASCII.

Handling interrupts. Devices in the system generate interrupts to inform the controller of error conditions, the completion of an operation, or other asynchronous events that require the controller's attention. The controller finds the device that generated the interrupt (if there is more than one instrument in the system), reads its status, and takes appropriate action.

Processing the Data

The third major task of a GPIB system controller is processing the data acquired from instruments. Often, a few important parameters must be extracted from a mass of raw acquired data. Again, the system controller takes over. A few instruments, such as the 492P Programmable Spectrum Analyzer and the 7854 Programmable Oscilloscope, can do

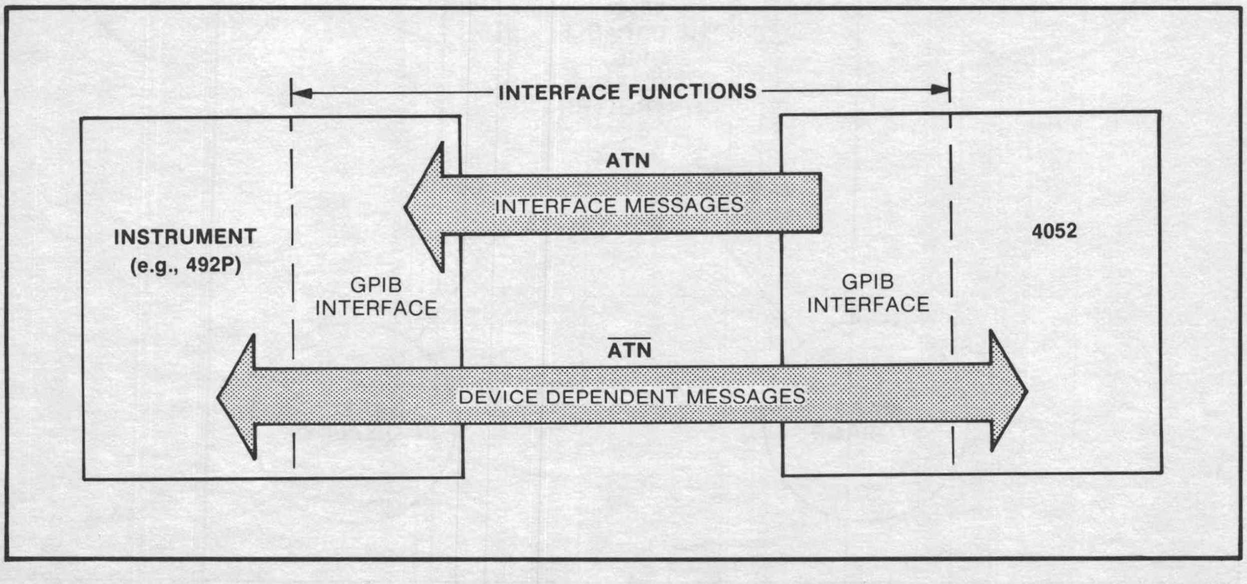


Fig. 1-3. *The controller sends interface messages with Attention (ATN) asserted. These messages control interface functions. Device dependent messages, sent with ATN unasserted, control instrument functions.*

some processing internally. But many can only send raw data to the controller, depending on it for processing.

This processing may involve simple operations, such as signal averaging or finding the amplitude of an acquired pulse. More advanced applications may require operations such as the fast Fourier transform (FFT) or convolution. Powerful high-speed minicomputers have made lengthy and complex calculations, once left to large mainframe computers, feasible even in a small GPIB system controller. With this power, the controller can set-up the instruments, acquire test data, and compute the desired parameters from the acquired data—all without human intervention.

Storing and Displaying the Data

Once data is acquired and processed, the controller is responsible for storing and/or displaying the results. Non-volatile mass storage, such as floppy disks or magnetic tape, provides a convenient means of logging data or results. In addition, the controller can generate graphic displays that make visual analysis of the data much easier. Hard copy (paper) output is also important for documentation of test results and displays. This

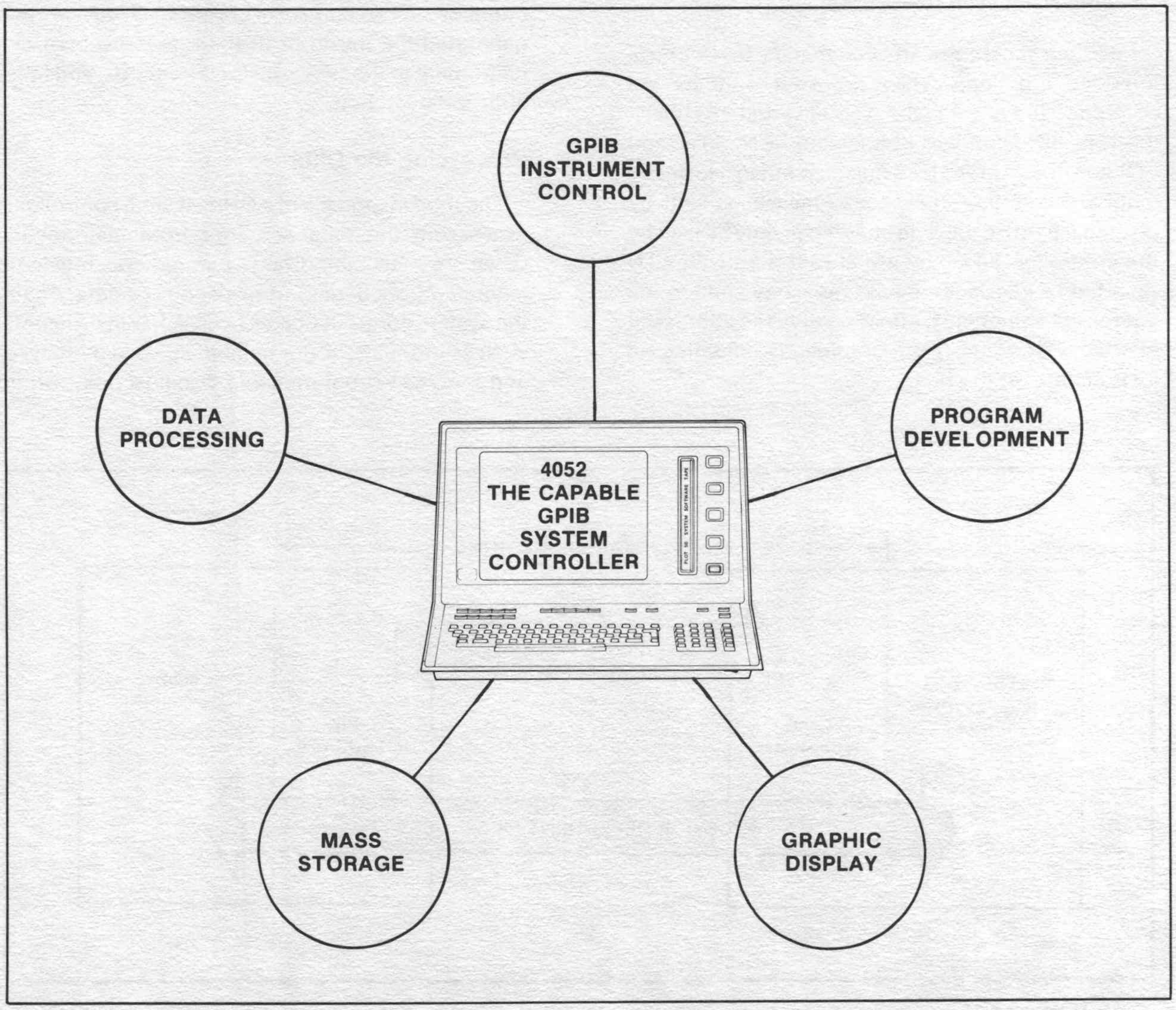


Fig. 1-4. *The 4052 is more than a GPIB-compatible desktop computer—it's a capable GPIB system controller.*

output may be provided by a hard copy unit that copies the screen contents or by a plotter.

GPIB Capability— More than GPIB Compatibility

An efficient, powerful GPIB system requires more than just a computer with a IEEE 488 interface—it requires a **capable** controller with the hardware, software, and peripherals to handle the tasks. Many a frustrated user has found that an IEEE-488 interface and a plug on the rear panel do not make a good GPIB controller. There is a considerable difference between GPIB compatibility and GPIB capability!

The Tektronix 4050-series Graphic Computing Systems have long been known as the leader in desktop graphics computers. But, they are also very capable GPIB controllers. The 4052 integrates a high-speed bit-slice microcomputer, a high-resolution graphics display, 300K bytes of magnetic tape mass storage, and a GPIB interface in a single compact unit (Fig. 1-4).

The 4052's powerful hardware is supported by an enhanced version of the popular, easy-to-learn 4050 BASIC originally implemented on the 4051. 4050 BASIC incorporates a flexible I/O structure that allows simple addressing of GPIB instruments and peripherals. It also includes extensions for signal processing, graphics, and GPIB control. Thus, the 4052 houses all the essential ingredients of a powerful, flexible, and **capable** GPIB system controller in a single cabinet.

Section 2—Configuring a 4052 GPIB System

The GPIB is a flexible interface—it can efficiently link many different types of instruments together to perform a variety of jobs. We have looked at the system controller's job and discussed some of the qualifications of a capable GPIB controller. But, choosing the right instruments and the right configuration for your system is also important. A clear definition of what you want the system to do and a basic understanding of the system components is the key.

Defining the System's Job

The first step in configuring a system is to define its job. Consider these questions:

1. What is the system's operating environment? Will it be performing repeated tests on a production line? If so, speed is probably a primary concern. On the other hand, accuracy is often more important in a research environment, where an operator sitting at a keyboard probably won't notice an extra second or two of delay.

2. Will the system need to generate test stimuli? If so, one or more signal sources will be required. And if the output of the source must be changed during a test, the generator(s) should be programmable.

3. Will the system acquire data? If the system is intended to make automated measurements, some type of data acquisition is a given. This acquisition could be as simple as a DC voltage measurement, or as complex as a high-speed transient waveform. The important points to consider here are the type of data to be acquired, number of data channels, and the GPIB capabilities required in the acquisition instrument. Also remember that the 4052 must be programmed to receive the acquired data. A variety of data formats are used, so be sure you know the specifics of how your instrument transmits its data. We'll talk more about this later.

4. Will the data need to be processed? If the acquired data requires processing, the controller or instrument must be capable of performing the necessary computations in the available time.

5. Will data or test results be logged to a peripheral device? In some cases, where data must be captured very quickly, data logging may be necessary. Data can be written to a peripheral device, sometimes without even passing through the 4052. Later, when the acquisition is complete, the 4052 can read and process data from the peripheral at a slower rate. It may even set up to log data from an acquisiton, initiate the acquisition, and process the data from the last acquisition while the next one is in progress.

Getting on the Right Path

These are some of the questions that need answers as you begin configuring your GPIB system. It's not an exhaustive list, but answering these questions will get you on the path to a clear definition of your system's job. And that's a big step toward a well-designed, efficient system.

Selecting System Components

With a clear definition of the system's purpose in mind, you can begin selecting the specific instruments to accomplish that purpose. This discussion focuses on the GPIB considerations of selecting components. Other required specifications will be determined by the application.

Is the instrument really programmable? Often, instruments that are described as "IEEE 488 programmable" in catalogs and sales brochures are actually only partially programmable. Some functions can only be set from the front panel or by internal controls. It's important to know which functions, if any, are NOT programmable when you are selecting instruments. For each component in the system, you should have a list of the functions that must be programmable. If, for example, you need a function generator, your list might include programmable frequency, phase, and symmetry. As you look for programmable function generators, look at the specifications carefully. Are these functions programmable? Don't assume that the functions you need will be programmable just because the brochure says the instrument is programmable.

In addition, most Tektronix programmable instruments provide a convenient query command that returns the current instrument settings to the controller. The settings are returned in a format that can be stored directly and transmitted back to the instrument as commands. Query commands are also included to return individual instrument settings or parameters.

This feature is especially valuable in interactive systems where the operator may make manual adjustments to the instrument through the front-panel controls. The operator may set-up the controls manually and send the settings to the controller by pressing a front-panel button or issuing a command from the controller.

How fast is the instrument? Speed can be an important factor in choosing GPIB system components, particularly for systems that are intended to perform high-speed tests in a production environment. The speed of a GPIB instrument is determined by three basic factors: the time required for acquisition, internal processing, and data transfer. If your system will be performing in an environment where speed is critical, take a careful look at the data transfer rate and other speed specifications of the instruments. Section 6 describes some techniques for estimating the performance of a GPIB system.

What interface functions are implemented? A device's GPIB interface provides the link between the GPIB and the programmable device functions. The IEEE 488 standard allows a designer to choose from a list of optional functions when implementing the device interface. These interface functions are defined in terms of the following "interface subsets:"

Source Handshake
Acceptor Handshake
Talker
Listener
Service Request
Remote/Local
Parallel Poll
Device Clear
Device Trigger
Controller

The instrument designer can choose to implement all, part, or none of each of these functions, as defined by the function subsets in the standard. You should find a list of the interface subsets in the specifications for any GPIB instrument. The list may sound strange until you realize that it's just a shorthand way of describing the device's interface functions. C0, for instance, says that an instrument has no capability as a controller. DT1 means that an instrument can be triggered to perform a designer-chosen function when it receives the group execute trigger interface message. A summary of the interface subsets is contained in appendix A.

As you select instruments for the system, keep these interface subsets in mind, but don't confuse them with the programmable functions of a device. The interface subsets only describe the capabilities of the device's GPIB interface, not the programmable functions of the device itself.

The same functions do not need to be implemented in all instruments. But, in many cases you won't be able to use the added capability of some instruments unless the controller or another instrument also implements it.

Consider, for example, a system that performs tests on a sample of units from a production line. Such a system is illustrated in Fig. 2-1. A DC5010 Programmable Counter/Timer counts the units coming off the production line and measures the frequency of the test response, an FG5010 Programmable Function Generator provides the test stimulus, and a 7912AD Programmable Digitizer captures the output waveform for the test.

The controller sets-up the function generator and digitizer to perform the test, setting them both to begin when they receive a group execute trigger message. When the DC5010 reaches the specified count, it interrupts the 4052. The 4052 sends a group execute trigger message, which starts the function generator output and starts the digitizer to capture the output waveform. When the test is complete, the counter is reset, and the process is repeated.

In order to implement this system, the device trigger interface function must be implemented on both the function generator and the digitizer. The moral of the story is—know what interface functions are required for your application. Then, check the interface function subset list to determine if the functions you need are implemented.

Addressing. The IEEE 488 standard defines the basic addressing scheme for GPIB instruments. However, it leaves several options open to the instrument designer, so it's also important to know the individual addressing requirements of the instruments you are considering.

All GPIB instruments have at least one primary address in the range 0-30. In most cases, this is the only address the user must be concerned with. The instrument actually has one address for talking (if it can talk) and one address for listening (if it can listen). But the controller automatically generates these "absolute" talk and listen addresses from the single primary address. When you use an output statement like PRINT, the controller adds 32 to the primary address to generate the absolute listen address of the instrument. When you use an INPUT statement, the controller adds 64 to the primary address to generate the absolute talk address. In most cases, this process is automatic, so the user need only remember the single primary address.

Some instruments also have one or more secondary addresses. This address selects a sub-function or part of the instrument to take part in the operation. The specific use of this secondary address is not defined in the standard, so manufacturers use it in several different ways. Again, the user specifies an address in the range 0-30, and the controller automatically adds 96 to this address to generate the absolute secondary address.

Addresses are usually set by a set of five switches inside the instrument or on the rear panel. These

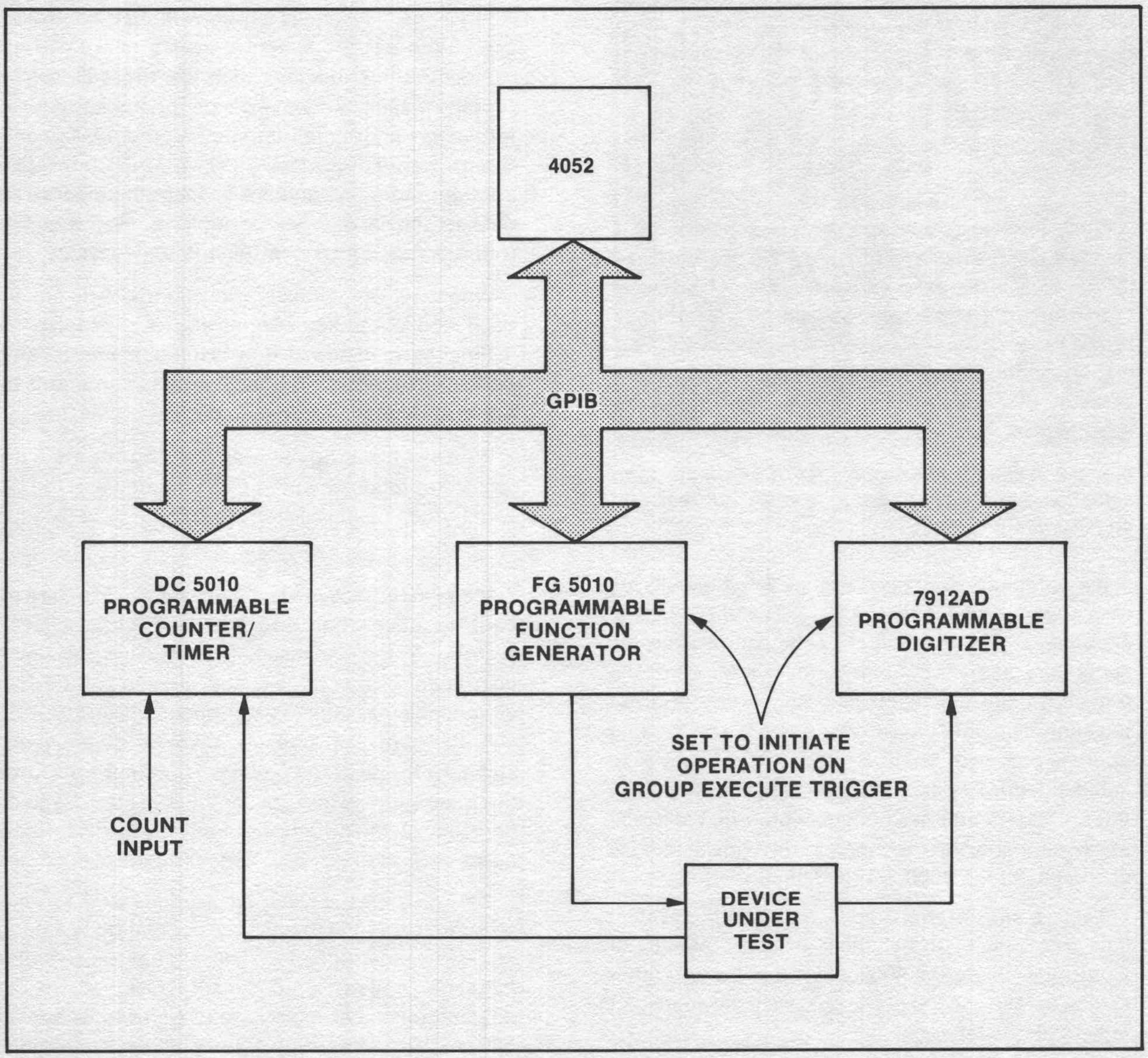

4052
GPIB
DC 5010 PROGRAMMABLE COUNTER/ TIMER
FG 5010 PROGRAMMABLE FUNCTION GENERATOR
7912AD PROGRAMMABLE DIGITIZER
SET TO INITIATE OPERATION ON GROUP EXECUTE TRIGGER
COUNT INPUT
DEVICE UNDER TEST

Fig. 2-1. *An example system using Group Execute Trigger (GET) to initiate an operation. Both the function generator and the digitizer must implement the DT1 interface subset (Device Trigger capability).*

switches allow you to set the address from 0 to 31, but there are some limitations. Some controllers, such as the 4052, reserve address zero for themselves, so you can't use address zero with these controllers. Also, 31 is not a valid address—it is used for the universal UNTalk and UNListen commands. Setting a device to address 31 effectively eliminates it from the bus since it can never be addressed. A typical set of address switches is shown in Fig. 2-2.

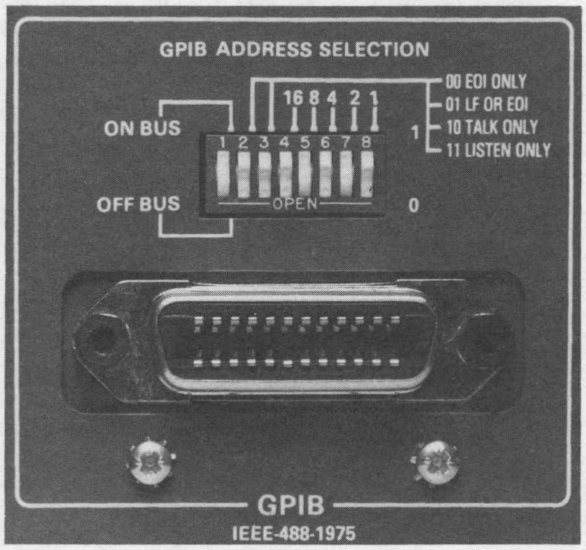


Fig. 2-2. *7854 Oscilloscope's GPIB connector and selection switches for setting primary address and communication mode.*

If a secondary address is required, it is usually set by a separate switch. In the 7912AD and 7612D, the secondary address switch sets the mainframe secondary address. The secondary address of the left plug-in is the mainframe secondary address plus one. The right plug-in address is the mainframe address plus two. So, to address the left plug-in to listen, the primary listen address is sent, followed by the secondary address of the left plug-in. We'll look at some specific examples of how this is accomplished in 4050 BASIC later.

Talkers and listeners. Instruments in the system can take one of three roles: Talker, Listener, or Controller. Since the 4052 will not allow any other device to take control of the bus, we'll only consider talkers and listeners.

At this point, it's important to understand what the IEEE 488 standard refers to as a "talk-only" or "listen-only" instrument. These terms refer to instruments that can be manually configured (usually with a switch) as permanent talkers or permanent listeners. When configured in this mode, the instruments do not need to be addressed by a controller. They are permanently addressed and they participate in every bus transaction. Other instruments may only be capable of talking or listening, but if they must be addressed by a controller, they are not considered "talk-only" or "listen-only" devices as defined by the standard.

Talk-only and listen-only instruments are useful when a small system is set-up without a controller. Often, the system simply consists of a talk-only acquisition instrument, such as the 468 Digital Storage Oscilloscope, and a peripheral configured for listen-only operation, such as the 4924 Digital Cartridge Tape Drive (Fig. 2-3). In this configuration, the acquisition instrument sends its data to the tape drive for logging. No other bus traffic occurs and a controller is unnecessary.

When talk-only or listen-only instruments are not used, the controller assigns the role of talker or listener to an instrument by issuing its talk or listen address, respectively. Unaddressed instruments do not participate in the transaction.

When choosing system components, it's important to know which instruments need to talk, which ones need to listen, and which ones need to do both.

Message format. Another important consideration when you are configuring a GPIB system is the message format used by each instrument. The syntax and coding of device-dependent messages is not specified in the IEEE 488 standard. As a result, there is no universal standard for message coding. This can be a source of frustration when programming the system, because of the widely different message formats used.

Tektronix has developed a codes and formats standard designed to enhance compatibility among its GPIB instruments. The standard specifies message coding and syntax designed to be unambiguous, correspond to those used by similar devices, and be as simple and obvious as possible. This standard makes programming a system of Tektronix GPIB instruments easier and simpler, because the messages for all instruments are similar and easy to remember. And since the commands consist of simple English-like mnemonics, programs are easier to read and understand.

Message terminators. Manufacturers also use different techniques to indicate the end of a message. Some instruments assert the EOI bus line when they are finished talking, others send a special character, such as line-feed. Again, the key is knowing what the instruments require. Using Tektronix instruments eliminates most of these problems, since they are designed to conform to the codes and formats standard, which specifies EOI as the message terminator. Most Tektronix instruments can also be configured to use the line-feed terminator when operated with other controllers.

Getting It Together

Now that you understand the capabilities and requirements for each instrument in your system, the job of actually configuring the system should be simple. This section provides a few guidelines to connecting the instruments together and setting the bus addresses.

Setting the bus address. The first step is setting the bus addresses for each instrument. Remember that every device must have a unique address. Valid primary addresses are 0-31, but don't use address zero—the 4052 reserves this address for itself. Also, selecting address 31 logically removes the device from the bus; it does not respond to any addresses, but remains both unlistened and untalked.

If you change the address switches after an instrument is powered-up, the address may not actually be updated until you return to local, re-initialize, or turn power off and back on. Check the instrument manuals for more details.

Since the 4052 POLL command allows you to sequentially poll instruments in any order, it is not necessary to arrange the addresses according to interrupt handling priority. As you set the addresses, write each one down for reference when writing programs.

Setting the message terminator. The message terminator on most instruments is selected with a switch on the rear panel or an internal strap. The most common delimiters are line feed and EOI. Tektronix controllers use EOI, but line feed option is available for compatibility with other controllers.

Cabling the instruments. The next step is cabling the instruments together. Up to 15 devices, connected by not more than 20 meters total cable length, can be interfaced to a single IEEE 488 bus. In some cases, more than 15 devices can be interfaced if they do not connect directly to the bus, but are interfaced through another device. For example, this scheme is used for programmable plug-ins housed in a 7612D or 7912AD Programmable Digitizer.

The system can be cabled in a star or linear configuration (Fig. 2-4). To maintain the bus electrical characteristics, a device load must be connected for each two meters of cable. Although devices are usually spaced no more than two meters apart, they can be separated farther if the required

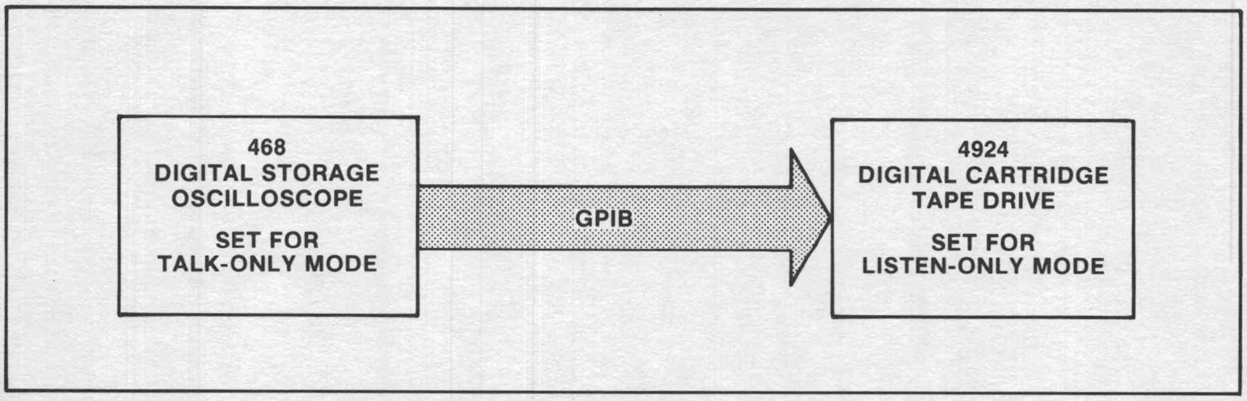


Fig. 2-3. *Some instruments can be manually set to permanent talker (talk-only mode) or permanent listener (listen-only mode). This allows small systems, such as the 468/4924 system shown here, to operate without a controller. These instruments may also be operated with a system controller.*

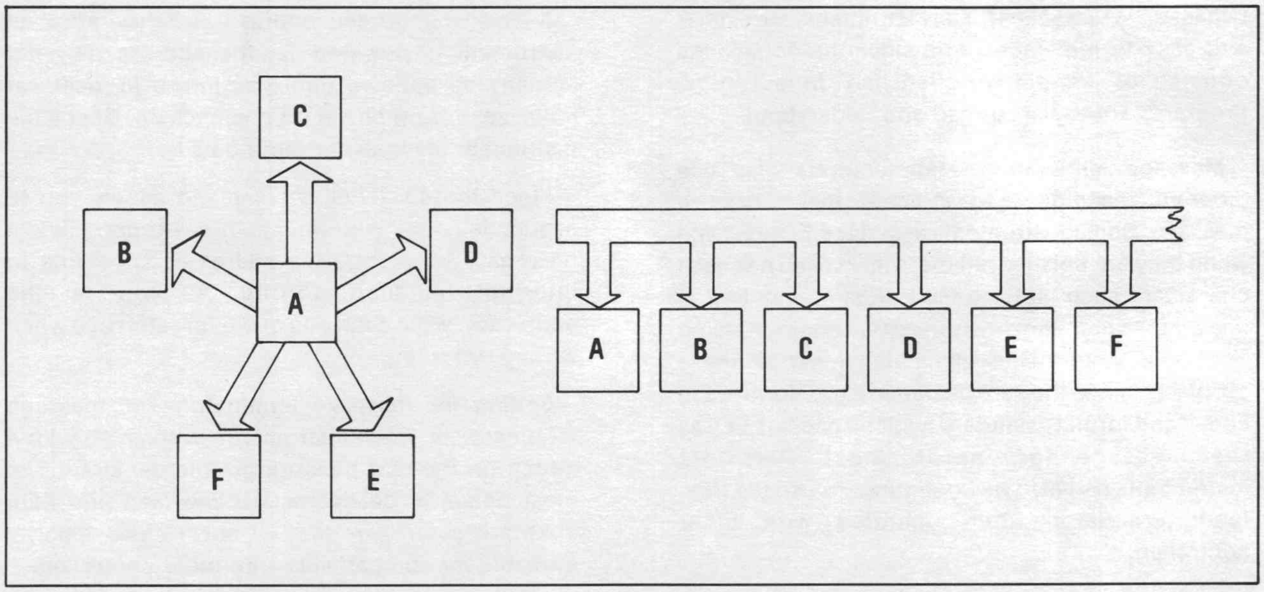


Fig. 2-4. *The GPIB system can be configured in either a star or linear manner.*

number of device loads are lumped at any point. If a single instrument is interfaced to a controller, the two-meters-per-instrument rule allows the controller and instrument to be separated four meters of cable.

Generally, at least two-thirds of the instruments on the bus should be powered-up for correct operation. In some cases, the bus will operate properly with fewer instruments powered-up. Check the standard for more details.

Section 3—Programming the 4052

Introduction to 4050 BASIC

The 4052 runs an enhanced version of 4050 BASIC. 4050 BASIC contains extensions in graphics, file system access, I/O operations, matrix operations, character string manipulation, high-level language interrupt handling, and operating system facilities. The 4052 implementation adds built-in matrix and binary program handling capabilities. Additional language extensions in file editing, signal processing, real-time control, and fast graphing are available using optional ROM packs.

If your controller will have to process waveforms or other array data, the signal processing ROM packs (4052R07 and 4052R08) will be particularly valuable. These two ROM packs provide 15 waveform and array processing functions, including differentiation, integration, maximum, minimum, and cross functions, fast Fourier transform, inverse Fourier transform, convolution, correlation, and others.

In process control and other real-time applications, the 4052R09 Real Time Clock ROM Pack is also very useful. It provides five time and date functions with elapsed time measurement and a programmable interrupt. All ROM pack routines are accessed with a simple CALL statement. More information on the ROM packs is provided in Section 5—"Processing and Displaying Data."

Although these extensions provide considerably more power than standard BASIC, most of the extensions are exercised through optional entries in the statements. This enhances compatibility with most other BASIC languages.

I/O Addressing in 4050 BASIC

4050 BASIC uses a powerful I/O addressing technique that handles all peripherals—internal and external—the same. For example, the same PRINT statement can be used to write data on the internal magnetic tape or to send ASCII data to an external GPIB-interfaced device. The only difference is the address specified in the statement. Let's look at a typical PRINT statement to see how this addressing technique works:

```
PRINT @33,12:"THIS IS ASCII DATA"
```

The keyword PRINT tells BASIC that data is to be output. The next two numbers are primary and secondary addresses. If they are not specified, the PRINT statement causes the data to be printed on the graphic display screen. When the addresses are included in the statement, the data is sent to the specified address. Addresses may be specified as constants, variables, or numeric expressions.

Each peripheral device in the system is assigned a primary address. For example, the above PRINT statement sends the ASCII string "THIS IS ASCII DATA" to the device at address 33, the internal magnetic tape drive. Primary addresses are assigned as shown in Table 3-1.

TABLE 3-1
PERIPHERAL DEVICE NUMBER ASSIGNMENTS

Device Number	Device
1-30	External devices on the GPIB
31	4052 Keyboard
32	4052 graphic display
33	4052 magnetic tape drive
34	DATA statement
35-36	Unassigned
37	Processor status
38-40	4050E01 ROM Expander
41	Left-most ROM slot
42-50	4050E01 ROM Expander
51	2nd-from-left ROM slot
52-60	4050E01 ROM Expander
61	3rd-from-left ROM slot
62-70	4050E01 ROM Expander
71	4th-from-left ROM slot
72-80	Unassigned

Each I/O statement in 4050 BASIC has a default address that refers to the internal peripheral usually accessed with that keyword. However, any valid primary address may be substituted for the default value. Table 3-2 lists the default addreses for each I/O statement.

The second number in the BASIC statement is a secondary address. Internal 4052 peripherals and some external peripherals use the secondary address to determine what type of I/O action is required. For example, if a KILL statement is executed, the internal tape drive is addressed by default. The default secondary address of 7 tells the tape drive that a KILL operation is being executed. The same operation could be executed using a PRINT statement by specifying the primary address of the tape drive, and the secondary address for the

KILL operation. Thus, these two statements are equivalent:

PRINT @33,7:n = KILL n

where **n** is the number of the file to be KILLed.

If a secondary address is specified in an I/O statement, it is sent in place of the default address.

TABLE 3-2
DEFAULT I/O ADDRESSES

BASIC Statement	Default I/O Address
APPEND	@33,4:
CLOSE	@33,2:
COPY	@32,10:
DRAW	@32,20:
FIND	@33,27:
FONT	@32,18:
GIN	@32,24:
HOME	@32,23:
INPUT	@31,13:
KILL	@33,7:
LIST	@32,19:
MARK	@33,28:
MOVE	@32,21:
OLD	@33,4:
PAGE	@32,22:
PRINT	@32,12:
RDRAW	@32,20:
READ	@34,14:
RMOVE	@32,21:
SAVE	@33,1:
SECRET	@37,29:
TLIST	@32,19:
WRITE	@33,15:

Some instruments, like the 7912AD and 7612D Programmable Digitizers share a single primary address among the mainframe and up to two programmable plug-ins installed in the mainframe. These instruments use the secondary address to select the mainframe or one of the plug-ins for involvement in an I/O operation. Others, like the TM 5000 series of programmable instruments, use a separate primary address for each instrument. Since the TM 5000 mainframe has no programmable functions, it is not assigned an address. When addressing instruments that use the secondary address for selecting a sub-function, simply specify the correct primary and secondary address for all I/O to the device.

4050 BASIC also simplifies addressing of external GPIB devices. A primary address in the range of 1 to 30 is used to address a device, whether to talk or listen. When an instrument is addressed to listen, 4050 BASIC adds 32 to the primary address to generate an absolute listen address. When an instrument is addressed to talk, 64 is added to the primary address. For example, if an instrument's primary address is set to 1, it listens at absolute address 33 and talks at absolute address 65. The 4052 automatically generates these absolute addresses from the primary address specified in the I/O statement.

The secondary address is also specified as a number in the range 0-30. 4050 BASIC adds 96 to this value to generate the absolute secondary address.

I/O Statements

The 4052's I/O statements can be separated into three levels as illustrated in Fig. 3-1. The highest level of statements perform special I/O functions that make programming easier and more efficient. For example, the DRAW statement makes graphics much simpler and faster than implementing the same function with PRINT statements.

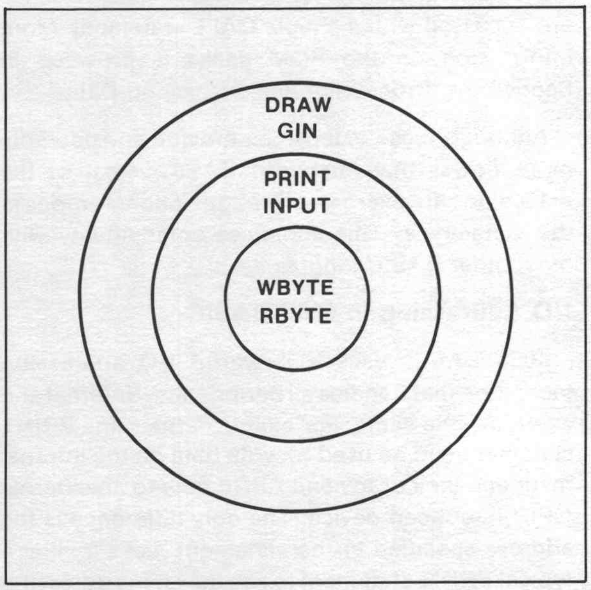


Fig. 3-1. *4052 I/O statements can be divided into three levels, from high-level statements like DRAW and GIN that implement special graphics I/O functions, to low-level statements like WYBTE and RBYTE that provide line-level control of the GPIB data bus.*

The next level of I/O statements are designed for simple operations such as sending or receiving ASCII data. The PRINT and INPUT statements provide some data formatting for ASCII input or output. The READ and WRITE statements perform a similar operation for machine-dependent binary input and output.

The lowest level of I/O statements are the RBYTE and WBYTE statements. These statements are intended to provide line-level control of the GPIB data bus at the expense of speed and increased complexity. RBYTE and WBYTE are exceptions to the standard addressing rules in 4050 BASIC. The absolute talk or listen address and the absolute secondary address (if required) must be specified, instead of the primary addresses used in higher level statements.

Bytes are tranferred in straight binary code. The programmer is responsible for checksums or other error checks. These statements give you full control of the GPIB data bus, and the ATN (Attention) and EOI (End or Identify) lines. The statements also allow you to set up a transfer between two instruments without passing the data through the 4052.

Refer to the 4050 Series Graphic Systems Reference Manual for a complete description of RBYTE and WBYTE.

Interrupt Handling

4050 BASIC also provides a simple high-level facility for handling service request (SRQ) and other interrupts. Statements are included to perform serial polls, transfer program control asynchronously on an interrupt condition, wait for an interrupt condition, or disable interrupts. Interrupt handling capabilities of 4050 BASIC are discussed in more detail in Section 4.

Section 4—Programming a 4052 GPIB System

Writing the programs that control a GPIB system is often the most time consuming and difficult part of building the system. But, with a clear definition of the system's purpose, carefully chosen components, and a powerful programming language like 4050 BASIC, the job is greatly simplified.

This section provides a guide for writing 4050 BASIC programs to control a GPIB system. The details of reading and writing commands and data, interrupt handling, and interface control are covered. A generous supply of sample programs are included. We'll also take a brief look at GPIB peripherals, such as floppy disk drives, tape drives, and plotters.

System Power-up

Power-up test. When it's time to power your system up, there are a few things you'll need to be ready for. First, remember that most programmable instruments automatically perform some kind of self-test procedure on power-up. The instruments usually won't respond to any front-panel or GPIB input until the power-up test is complete—all you can do is wait. The time required to complete this procedure varies from milliseconds to several seconds.

If all goes well in the test, the instrument powers-up normally. Otherwise, errors are usually reported on the front panel and by setting the status byte to indicate the error. When the self-test is complete or errors are detected, the instrument asserts the GPIB SRQ line to tell the controller that its status can be read.

Power-up SRQ. The 4052 hears the SRQ, but since the SRQ line is shared among all instruments on the bus, it can't tell which instrument(s) are asserting it. Nor can it tell at this point whether the instruments completed their power-up tests normally, or if errors were detected. The solution is to read the status byte from each instrument. This accomplishes two things: First, it tells the 4052 if the instruments powered-up normally, and if they didn't, what's wrong. This information can be passed on to the system operator via the controller's display. Second, reading the status byte clears the SRQ.

If the GPIB system is powered up before the 4052 has been told what to do with SRQ's, it prints this error message:

```
NO SRQ ON UNIT - MESSAGE NUMBER 43
```

This error isn't serious, it's just warning you that one or more instruments on the bus are asserting SRQ and that the 4052 doesn't know what to do about it. Again, the solution is simple—use the POLL command to read the status byte from each instrument. The command format is:

```
POLL X,Y;<primary address,[secondary address]>
```

(The secondary address is optional—use only when required.)

In the simplest case, where a single instrument is connected to the 4052 and set for bus address 1, the command is:

```
POLL X,Y;1
```

The instrument's status byte is returned in the variable Y and a 1 is returned in X, indicating that the first device polled was asserting SRQ. Several addresses may be specified in a single POLL command by separating each address with a semicolon. Secondary addresses may also be included by separating them from the primary address with a comma. For example, the statement:

```
POLL X,Y;1,1;2;3
```

first polls the device at address 1 with secondary address 1. This might be a programmable plug-in installed in a programmable mainframe. If this device is not requesting service, the next device in the list (address 2) is polled. If this one isn't asserting SRQ, the last instrument (address 3) is polled.

When the instrument that is asserting SRQ is found, its position in the list of addresses is returned in the first variable (X in our example). The status byte from this instrument is returned in the second variable (Y). The polling stops with the first instrument that is found asserting SRQ. If several instruments are asserting SRQ, the POLL statement must be executed once for each device asserting SRQ.

To illustrate, assume that a system contains the following instruments set for the addresses shown:

Instrument	Primary Address
492P Spectrum Analyzer	2
468 Oscilloscope	3
CG551AP Calibration Generator	15
DC5010 Counter/Timer	20

When the system is powered up, all four instruments assert SRQ. The program shown in Fig. 4-1 polls the instruments, and prints the status bytes on the 4052 screen.

```
10 D(1)=2
20 D(2)=3
30 D(3)=15
40 D(4)=20
50 POLL X,Y;2;3;15;20
60 IF X=0 THEN 90
70 PRINT "SRQ FROM DEVICE NUMBER ";D(X);" STATUS: ";Y
80 GOTO 50
90  .
    .
    .
    .
```

Fig. 4-1. *A simple program to poll instruments and print the status bytes on the 4052 screen.*

Lines 10 through 40 load array D with the primary addresses of each instrument in the order they are listed in the POLL command. Line 50 begins polling the instruments, starting with the 492P, at address 2. Since it is asserting SRQ, the polling process stops, the 492P status byte is returned in Y, and the position of the 492P in the address list is returned in X.

Then, line 70 prints the address and status of the device found asserting SRQ. Notice that X does not contain the address—it contains the **position** of the address in the list. The value of X determines which element of array D will be printed. If the first instrument is asserting SRQ, a one is returned in X, and the address of the first instrument from array D will be printed.

Finally, line 50 transfers control back to the POLL statement. Since the 492P SRQ has been cleared, this time the POLL command goes on to the next device in the address list, the 468. Its status byte is returned in Y, and a 2 is returned in X, indicating that the second device in the list was found asserting SRQ. This process is repeated until none of the devices in the list are found asserting SRQ. Then, the POLL command returns a zero in X and the IF statement in line 60 transfers control out of the loop.

SRQ interrupts can also occur for a number of reasons other than power-up. We'll discuss interrupt handling in more detail later.

Device Dependent Messages

Device dependent messages comprise the vocabulary of a GPIB instrument. The content and format of these messages is not defined by the IEEE 488 standard; it is determined by the instrument designer to suit the needs of the particular instrument. Device dependent messages may include queries that return instrument settings or acquired data, commands that control instrument settings, or data such as waveforms or measurement results.

This section describes the format of device-dependent messages for Tektronix instruments as well as the techniques for transmitting and receiving these messages with the 4052.

Device dependent message I/O. Regardless of the message content, coding, or format, the basic process of transferring a device-dependent message is the same. The message is always transferred from a device addressed as a talker to one or more devices addressed as a listener. The process is illustrated in Fig. 4-2.

First, the talker and listener(s) must be assigned. The controller asserts the ATN line and puts the appropriate address on the bus. If a secondary address is required, it directly follows the primary address. If, for example, the 4052 is sending a message to a device, the listen address of that device is placed on the bus. The 4052 automatically assumes the role of talker when outputting a message and that of a listener when receiving a message.

When the addressing sequence is complete, the 4052 releases attention and puts the first byte of the message on the bus. The bytes are transferred one at a time at the rate of the slowest listener until all bytes are sent.

Then, the 4052 asserts ATN again and sends the universal UNTalk or UNListen command. If the message was sent from an instrument to the 4052, UNT is issued. If the message was sent from the

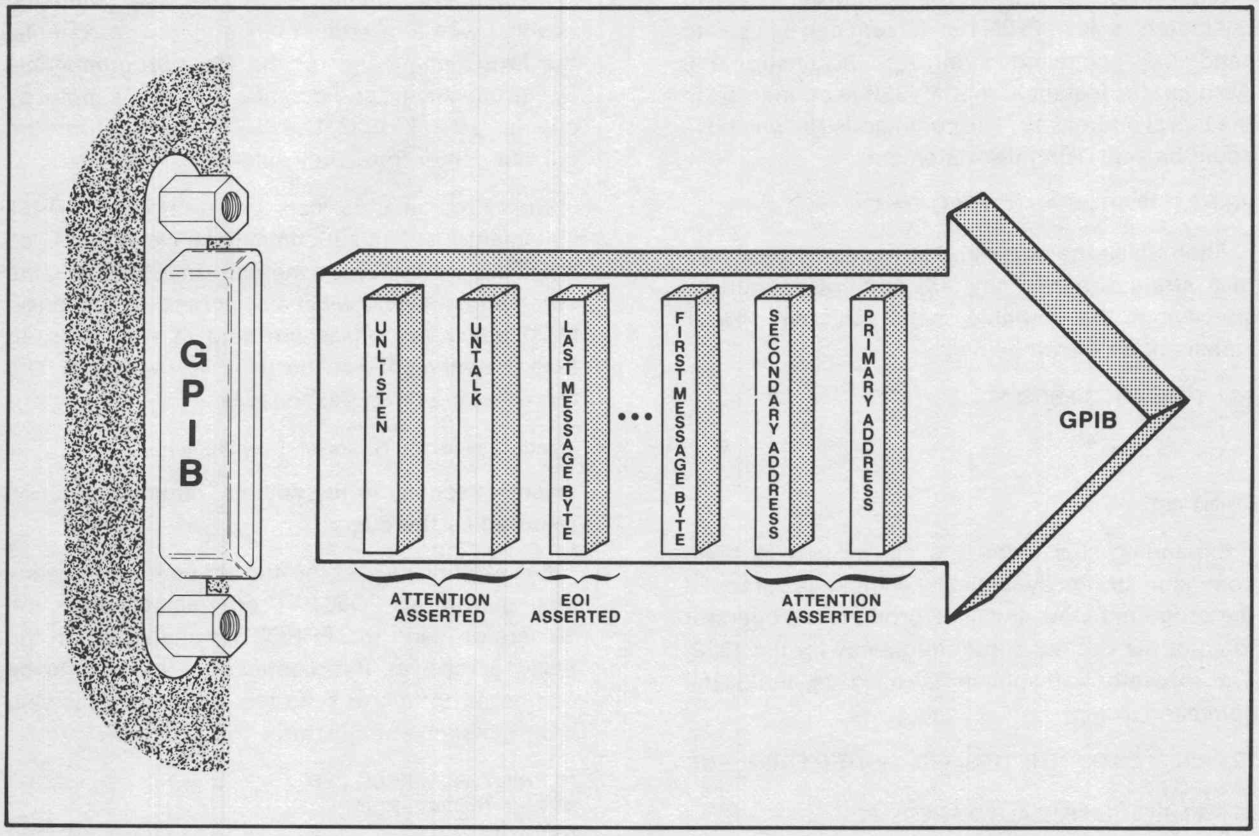


Fig. 4-2. *The basic process of transferring device-dependent messages is the same, regardless of the content and format. A primary address is sent first, followed by a secondary address (if required). Then, the data bytes are sent, followed by the UNTalk and UNListen commands.*

4052 to a listening instrument, UNL is issued. This clears the bus and leaves it ready for the next transfer.

Fortunately, most of the mechanics of transferring messages is transparent to the user; the 4052 takes care of it. In the special cases, low-level commands are provided that allow you to control bus activity more directly.

Set commands. Device dependent messages that set instrument operating modes or parameters are called set commands. In Tektronix GPIB instruments, these commands take the following form:

```
<header>[<space><arguments>][<semicolon>]
```

The header is a mnemonic or keyword that identifies the command. If the command requires arguments, they are separated from the header by a space. The arguments specify the values or parameters required by the command. For example, the TIME AUTO command in the 492P Programmable Spectrum Analyzer sets the time/division to automatic mode. TIME is the command header and AUTO is an argument. Using the MAN argument (TIME MAN), sets the time/division to manual mode. Other commands, such as FREQ 1MHZ (set center frequency to one megahertz), require numeric arguments. Still others require no aruments at all, such as SIGSWP (single sweep).

Multiple set commands can be grouped together and sent as a single message by separating the commands with semicolons. For example:

```
TIME AUTO;FREQ 1MHZ
```

combines the TIME and FREQ commands into a single message.

Since these commands are composed of ASCII characters, a 4052 PRINT statement can be used to send the command string to the instrument. Assume, for instance, that a 492P is connected to the bus at address 10. The commands shown above could be sent using the statement:

```
PRINT @10:"TIME AUTO;FREQ 1MHZ"
```

Alternately, the command string could be stored in a string variable, say A$, and transmitted by specifying the variable name in the PRINT statement as shown below.

```
A$="TIME AUTO;FREQ 1MHZ"
  .
  .
  .
PRINT @10:A$
```

Expanding this technique allows you to build command strings dynamically within a program. In the program below, the 4052 prompts the operator to enter the desired center frequency for the 492P. The operator's response is used to build the command string.

```
10 PRINT "ENTER THE DESIRED CENTER FREQUENCY :";
20 INPUT F$
30 PRINT @10:"TIME AUTO;FREQ ";F$
```

Line 10 prompts the user for the center frequency and line 20 stores the response in the variable F$. Then, line 30 appends F$ to the end of the command string and sends the completed string to the instrument. Expanding this concept with a few simple statements would allow the program to check the response for validity, report errors to the operator, and request valid input.

A numeric variable could also be used in the above example in place of the string variable F$. When a numeric variable is used in the PRINT statement, the value is automatically converted to a string of ASCII digits before it is transmitted. The original variable is unaffected. Numeric variables can also be used with INPUT statements. The conversion from ASCII input to internal numeric format is automatic.

Query commands. Query commands are device-dependent messages that cause the instrument to return information about its settings or operation. The form of the query commands is:

```
<header><question mark>[<semicolon>]
```

Many query commands are simply set command headers with a question mark added. For example, the FREQ command for the 492P Programmable Spectrum Analyzer becomes a query simply by changing it to FREQ?. The FREQ? query returns the current center frequency setting.

Query commands can be sent in a PRINT statement just like set commands. However, since the instrument returns a message in response to the command, the 4052 must also accept the response. Most query responses are sent in ASCII, so an INPUT statement can be used to receive it. The format of the query response is:

```
<header><space><response>[<semicolon>]
```

where: response is the setting, value, or function returned by the query.

For example, to get the current center frequency setting from an FG5010 Programmable Function Generator, send the FREQ? command. When the FG5010 receives this command it expects to be addressed to talk to send the reply. The following program segment illustrates this process.

```
10 PRINT @24:"FREQ 2E6"
20 PRINT @24:"FREQ?"
30 INPUT @24:F$
```

Line 10 sets the output frequency to 2 megahertz. Then, line 20 sends the FREQ? query. Line 30 gets the complete reply (header and value) and puts it in the string variable F$. In this example, F$ would contain the string:

```
FREQ +2.0E+6;
```

If you only want the numeric value from the response, simply specify a numeric variable in the INPUT statement, instead of a string variable. The header (FREQ) will be ignored and the value will be returned in the specified variable. In the previous example, if a numeric variable had been specified, say F0, the FREQ header would be ignored and the value of the current frequency returned in F0.

Set and query commands may be grouped together in a single message by separating the commands with semicolons. For example, the two PRINT statements in the previous example could be condensed into a single statement by combining the set and query commands as follows:

```
10 PRINT @24:"FREQ 2.0E+6;FREQ?"
```

In most instruments several queries may be included in a single message. However, the specific rules for multiple commands in a message vary slightly among instruments, so refer to the Operator's or Programmer's Manuals for your instruments.

Sending ASCII data. The controller may send waveform or other data to instruments for processing or display. The format of this data depends on the instrument, but many accept it in ASCII-coded decimal numbers. For example, the 492P can accept ASCII waveform data using the CURVE command. Several options are available with this command, but for the sake of example, consider its simplest form:

```
CURVE <data value><comma or space><data value>
<comma or space>...
```

A typical CURVE command might be:

```
CURVE 27,28,29,31,33,36,39,44,...
```

The 4052 can transmit the CURVE command and ASCII data using a simple PRINT statement such as

```
PRINT @1:"CURVE ";A;
```

When this statement is executed, the 4052 addresses the 492P and sends the CURVE command header. Then, it begins sending the contents of array A. Transmission starts with the first element, A(1), and ends with the last. The semicolon on the end of the statement suppresses the extra spaces added by the 4052 between array elements. It is not required, but it reduces the number of bytes sent by eliminating the extra spaces, and thus, speeds up the transfer.

Array elements may not be sent using separate PRINT statements because each statement asserts EOI with the last byte transmitted, so each value is sent as a separate message. The instrument requires that all data points be sent in a single message.

The data can also be transmitted from a string array by substituting the string variable name for the numeric variable and deleting the trailing semicolon, as shown below.

```
PRINT @1:"CURVE ";A$
```

When the data is transmitted from a string variable, the contents of the string are transmitted without modification. Processing the data is more difficult when it is stored in a string, since individual data elements cannot be easily accessed. But, transmitting data from ASCII strings is faster than transmitting from a numeric variable because the 4052 does not have to perform any data conversion. For simple waveform storage, when no processing is required, storage in string variables is probably best. When processing is required, the data should be stored in numeric variables.

Reading ASCII data. Data can also be received from an instrument in ASCII-coded decimal numbers. The 4052 INPUT statement accepts data from the instrument and stores it either in numeric variables or string variables. For example, the statement

```
INPUT @10:A
```

reads a single ASCII number and stores it in the variable A. If A was previously dimensioned as an array, the 4052 attempts to read one number for each element in the array. When more than one number is received with a single INPUT statement, individual numbers must be delimited by a non-numeric character (valid numeric characters are 0-9, +, −, and in scientific notation, E). Most instruments delimit each data value with a space or comma.

Also, since non-numeric characters are ignored in numeric input, this can be a handy way to strip unwanted data headers off and store only the numbers. For example, when the 492P sends a waveform in response to a CURVE? query, a waveform header preceeds the data. A typical CURVE? query response is shown below.

```
CURVE CRVID:FULL,27,28,29,31,...
```

If a numeric array is specified in the INPUT statement, the 4052 ignores the ASCII header characters and begins storing data with the first waveform value (27). Each value is stored in successive array elements.

ASCII data can also be read into a string variable by specifying a string variable in the INPUT statement. When data is read into a string, all ASCII characters are stored exactly as sent. Reading data into a string is faster than reading into numeric variables because no data sorting or conversion is necessary. And, since query responses can be directly transmitted back to the instrument as a command, string storage is an efficient means of saving instrument parameters, waveforms, or settings for sending back to the instrument later.

Using alternate delimiters on INPUT. When multiple variables or an array is received with a single INPUT statement, the individual elements must be properly delimited. For numeric input to multiple variables or arrays, the delimiter is simple. Any non-numeric character (characters other than 0-9, +, -, and E in scientific notation) is a valid delimiter.

INPUT to string variables isn't always so simple. The problem is that the only valid delimiters for string input are EOI, carriage return, or an alternate delimiter.

String input can be broken into parts using the alternate input delimiter feature in the 4052. If a percent sign (%) is specified in place of the at sign (@) in the I/O address for the INPUT statement, the 4052 uses a previously specified ASCII character for a delimiter. The delimiter character is defined by modifying the processor status parameters with a PRINT statement.

```
PRINT @37,0:n1,n2,n3
```

The first number (n1) specifies the ASCII decimal code for the INPUT delimiter. It must be in the range of 0-255. If, for example, 65 is specified, the ASCII letter "A" delimits string INPUT just as non-numeric characters delimit numeric input. But remember, the alternate delimiter is only used when the INPUT %N form is used.

The second number (n2) specifies the ASCII decimal code for the end-of-file character. This value must be in the range 0-255, and it must be specified, whether or not it is changed from the default (255).

The third number (n3) specifies the ASCII decimal code for the character that will be deleted from incoming ASCII strings. If, for example, 67 is specified, all upper case C's are deleted from the input. This value must be specified whether or not the default (255) is changed.

Once the alternate delimiter is specified, an INPUT %N: statement can be used to read data up to the delimiter character. If, for example, you want to read ASCII CURVE data from a 492P and store the waveform identifier separately from the data, the routine shown in Fig. 4-3 will do the job.

```
10 REM *** SPECIFY ALTERNATE DELIMITER ***
20 PRINT @37,0:44,255,255
30 REM *** NOW READ THE DATA ***
40 PRINT @10:"CURVE?"
50 INPUT %10:W$,A
```

Fig. 4-3. *A program to read ASCII data from the 492P using alternate delimiters.*

Line 20 changes the INPUT delimiter to a comma (ASCII code 44). Then, line 40 sends the CURVE? query and line 50 begins by reading the curve identifier into W$. The identifier is separated from the first data value by a comma, so data storage in W$ stops at the end of the header. The remaining data values are stored in array A.

Binary waveform data format. The 4052 can also transmit waveform data in binary as required by some instruments. For example, the 492P can accept waveform data in binary as well as ASCII. Binary data transmission is slightly more complex than ASCII transmission, but it's considerably faster.

The Tektronix codes and formats standard specifies two formats for binary data transmission. Some instruments, like the 492P, can accept data in either format. Others, like the 7912AD Programmable Digitizer, require one format. Check your instrument Operators or Programming manual for the required format.

The first format is called the "end block binary" format. It is simple, but has no provision for error checking. The format is:

```
@<data value><data value>...
```

@ is the ASCII code for the "@" character. This tells the instrument that binary data in the end block binary format follows.

DATA VALUE is an 8-bit binary number. If the instrument requires 16-bit values, two bytes are sent for each value, most significant byte first. EOI is asserted with the last byte in either case.

The second format, called the block binary format, is more complex since it includes a byte count and checksum for error checking. This format is:

```
%<byte count><data value><data value>...<checksum>
```

% is the ASCII code for a "%" character. This tells the instrument that a binary block with error checking follows.

BYTE COUNT is a 16-bit binary number that indicates the number of bytes remaining in the message, including the checksum. The byte count is sent as two bytes, most significant byte first.

DATA VALUE is an 8-bit binary number. If the instrument requires 16-bit values, two bytes are sent for each value, most significant byte first.

CHECKSUM is an eight-bit binary number that is the 2's complement of the modulo 256-sum of all preceding bytes except the first (%). That is, the eight-bit sum of the preceding bytes, ignoring the carry. If the receiver computes a modulo-256 sum of all the bytes except the percent sign, but including the checksum, the result should be zero. Thus, the checksum provides an error check for the binary block transmission.

Sending binary data. Binary data can't be transmitted with the PRINT statement because the 4052 converts all data in a PRINT statement to ASCII before sending it. However, 4050 BASIC provides a low-level I/O statement that gives the programmer line-level control of the GPIB: WBYTE (Write Byte). WBYTE can send any byte on the GPIB with or without attention (ATN) or EOI asserted. In its general form, the syntax of the WBYTE statement is:

```
WBYTE @<attention byte>...:<device-dependent bytes>
```

When WBYTE is used to send device-dependent messages, the syntax is:

```
WBYTE @<listen address>[,<secondary address>...]:<bytes>...
```

All bytes sent before the colon are sent with attention asserted; all bytes after the colon are sent with attention unasserted. Any byte in the range +255 through -255 may be sent. Positive values are sent with EOI not asserted; negative values cause the byte to be sent with EOI asserted.

The addresses specified in the WBYTE statement must be absolute physical addresses, instead of the peripheral device number. The listen address for an instrument is simply its peripheral device number plus 32, the talk address is the peripheral device number plus 64, and the secondary address is the peripheral device number plus 96. For example, to address an instrument set for primary address 1 and secondary address 1, use the statement:

```
WBYTE @33,97:<data bytes>...
```

This statement tells the 4052 to assert attention and send the listen address (32+1=33). Then, with attention still asserted, it sends the secondary address (96+1=97). After releasing attention, the device-dependent data bytes following the colon are sent.

As an example, the following program segment addresses the 492P to listen and sends a simple binary block stored in array A.

```
10 WBYTE @33:ASC @",A,-255
20 WBYTE @63:
```

Line 10 of the program first sends the 492P listen address (33) with attention asserted. Then, with attention unasserted, the ASCII code for "@" is sent, followed by the data array (A). The -255 byte is sent as a message terminator. The 492P buffers this byte, but it is not included in the waveform data. The last byte of the waveform data could also be negated and used as a message terminator, eliminating the need for the -255 byte. Line 20 uses the general form of WBYTE to send an interface message, UNListen (63) with attention asserted. This message tells all addressed listeners to stop listening.

If an ASCII command header, such as CURVE or LOAD, is required, the header is sent before the @ or % characters.

Generating the byte count for block binary. The block binary format includes a byte count for error detection. The byte count indicates the number of bytes remaining in the message, including the checksum.

To compute the byte count, simply add 1 to the size of the data array to account for the checksum. Since the byte count is often too large a number to be sent in a single byte (255 is the maximum), it must be divided into two bytes before being transmitted. The two bytes are appended by the receiver to form a 16-bit number. If the byte count is smaller than 255, the high-order byte is set to zero.

Dividing the byte count into two bytes requires only two lines of 4050 BASIC. The following program segment takes a byte count value stored in B2 and converts it to an equivalent two-byte value in B0 and B1.

```
10 B0=INT(B2/256)
20 B1=B2-B0*256
```

To see how this routine works, consider an example. A byte count of 515 is represented in binary as:

```
1000000011
```

Since this number is expressed in ten bits, it must be divided into two bytes. Line 10 of the program divides the byte count by 256, truncates the result to an integer, and stores it in B0. In this case, B0 contains a 2. Since the value of the least significant bit in B0 is 256, the 2 in B0 actually represents a value of 512.

Line 20 of the program subtracts the value of B0 (256 times B0) from the byte count, and stores the remainder (3) in B1. Thus, the byte count is represented in two bytes as:

```
   B0         B1
00000010  00000011
```

With the byte count separated into two bytes, it can be transmitted with a WBYTE statement.

Generating the checksum. The block binary format also includes a checksum byte for error checking. The checksum is computed by taking the modulo-256 sum of all bytes in the block except the % character and the checksum. The result is converted to it's 2's complement before transmission. When the instrument receives the bytes, it computes the modulo 256 sum of the bytes and adds the checksum. If the result is zero, the transmission is assumed to be correct.

The following program segment computes a checksum for binary data stored in array A.

```
10 C0=SUM(A)+B0+B1
20 C0=256-(C0-256*INT(C0/256))
```

Line 10 computes a sum of all the data values in array A and adds the byte count, B0 and B1. Then, line 20 converts this sum to a modulo-256 sum by subtracting the largest integer multiple of 256 that leaves a positive remainder. This remainder is subtracted from 256 to produce the checksum byte.

Sending block binary data. With the byte count and checksum computed, the complete binary block can be sent with two simple statements. The program in Fig. 4-4 shows the process of computing the byte count and checksum, transmitting the binary block, and unaddressing the instrument.

```
10 B2=N+1
20 B0=INT(B2/256)
30 B1=B2-B0*256
40 C0=SUM(A)+B0+B1
50 C0=256-(C0-256*INT(C0/256))
60 WBYTE @34:ASC "%",B0,B1,A,C0
70 WBYTE @63:
```

Fig. 4-4. *A few simple 4050 BASIC statements calculate the byte count and checksum, and transmit the binary block.*

Lines 10 and 20 split the byte count in B2 into two bytes and lines 40 and 50 compute the checksum. Then, line 60 transmits the binary block. The block begins with the ASCII code for "%", followed by the byte count, B0 and B1. Next comes the data array (A) and the checksum (C0). Since the checksum byte will always be negative, EOI is asserted when it is sent. Line 70 sends the UNListen message to unaddress the instrument.

Reading binary data. Many instruments send waveform or other data to the controller in binary. Tektronix instruments send binary data in either the block binary or end block binary format previously described. In either case, the RBYTE statement in 4050 BASIC is used to read the data.

The syntax of the RBYTE statement is:

```
RBYTE <numeric variable>,[<numeric variable>]...
```

RBYTE simply accepts bytes from the talker and assigns them to the variables in the list. If an array variable is specified in the list, the 4052 reads data from the bus and begins filling the array starting with element 1. It continues to read data into the array until it is full. If EOI is asserted with a byte, the value is negated before it is stored in the variable and the transfer is terminated.

Before RBYTE can receive data from an instrument, the instrument must be told what to say. This is usually accomplished with a PRINT statement (e.g. PRINT @1:"CURVE?"). Then, it must be addressed to talk. When receiving ASCII data with the INPUT statement, this process is automatic. When receiving binary data with RBYTE, the talk address must be manually sent using the WBYTE statement.

The program in Fig. 4-5 illustrates this process by reading a waveform from the 492P.

```
10 PRINT @37,0:37,255,255
20 PRINT @1:"WFMPRE ENC:BIN;CURVE?"
30 INPUT %1:H$
40 WBYTE @65:
50 RBYTE B0,B1,A,C0
60 WBYTE @95:
```

Fig. 4-5. *A program to read binary data from the 492P using alternate delimiters.*

Line 10 changes an internal status flag in the 4052. It tells the 4052 to delimit INPUT strings with the character whose ASCII code is 37 (%). Then, line 20 sends a message to the 492P that tells it to transmit data in binary and requests the data. Since the ASCII waveform header comes first, the INPUT statement in line 30 reads this header. The INPUT operation stops when the alternate delimiter selected by line 10 (%) is reached.

The first character in the binary block is a "%", so the INPUT operation stops reading at that point. Line 40 addresses the 492P to talk again, and line 50 gets the waveform data. The first two bytes are the byte count. They are stored in B0 and B1. The data is read into the previously dimensioned array A. The last byte is a checksum, which is stored in C0. Finally, line 60 sends the UNTalk message to unaddress the 492P.

Sending Interface Messages

Sometimes it may be necessary to send an interface message or sequence of messages that are not implemented in a high-level 4050 BASIC statement. The WBYTE statement allows you to send any byte on the GPIB with or without attention asserted. Using WBYTE a program can send any interface messages that are required.

A simple example is sending the device clear message. This universal command resets the interface functions of all devices on the bus. The WBYTE statement shown below sends a decimal 20 byte. Since the byte is before the colon, attention is asserted and the byte is interpreted as the device clear interface message, instead of device-dependent data.

```
WBYTE @20:
```

An example of using WBYTE to perform a serial poll is shown is **Polling instruments that are not asserting SRQ**.

Transfers Among GPIB Devices

The 4052 can also set up a transfer between two or more devices on the bus without involving itself. For example, you might want to transfer data from a tape drive to a plotter directly. If the data is stored in a format that the plotter understands, the 4052 does not need to be involved and the tranfer may be faster without its involvement.

The WBYTE statement allows you to assign a talker and one or more listener and then relinquish control of the bus to the talker, waiting for the EOI line to indicate that the transfer is complete. The following example illustrates how this is accomplished.

```
150 ON EOI THEN 180
160 WBYTE %70,109,52,35:
170 WAIT
180 WBYTE @63,95:
```

When line 150 is executed, the 4052 is told to transfer control to line 180 when EOI is asserted (the ON statement will be described more fully in the next section). Line 160 sends primary talk address 70 followed by secondary address 109. The primary address assigns this device as a talker. In this case, the secondary address tells the peripheral that it should transmit ASCII data in the upcoming transfer. Next, primary listen address 52 and primary listen address 35 are sent to assign these devices as listeners. The percent sign (%) in the WBYTE statement tells the 4052 to get off the bus and let the assigned talker take over when the ATN line is released.

At this point, the talker takes over the bus and starts sending data to the two listeners. Meantime, the 4052 waits at line 170 for the transfer to complete. The talker must assert EOI with the last byte of the message to signal the 4052 that the transfer is complete. When EOI is asserted, program control is transferred to line 180 and the 4052 assumes control of the bus again. It asserts ATN and sends the UNListen (63) and UNTalk (95) messages.

When specifying primary and secondary addresses in a WBYTE statement, only one device may be assigned as a talker, but up to 14 devices may be assigned as listeners (there can only be 15 devices on the bus). Secondary addresses and interface messages may also be sent in the message as necessary.

Interrupts and Instrument Status

Interrupt conditions. Sometimes special conditions, called interrupts, occur that cause the 4052 to temporarily suspend normal program flow. When one of these conditions occurs, the 4052 suspends execution of the current program and jumps to a special user-written routine that handles the interrupt. When the routine is complete, normal program execution resumes at the point it was interrupted.

The four interrupt conditions are:

- SRQ (Service Request) from an external GPIB device.
- EOI (End Or Identify) from an external GPIB device.
- EOF (End Of File) from the internal magnetic tape unit.
- SIZE errors cause by numeric underflow or overflow in a program.

Each of these conditions must be handled by a user-written BASIC routine (the EOI condition can be ignored when the 4052 is involved in the GPIB transfers). If they occur and no service routine is included in the current BASIC program, an error message is printed and the program is terminated.

The starting line number of each interrupt service routine is specified with an ON statement. This statement tells the 4052 where to transfer control when a specified interrupt occurs. One ON statement must be included for each interrupt service routine. The syntax of the ON statement is:

$$\text{ON} \left\{ \begin{array}{l} \text{EOF (0)} \\ \text{SIZE} \\ \text{EOI} \\ \text{SRQ} \end{array} \right\} \text{THEN line number}$$

When an ON statement is executed, the 4052 establishes a link between the specified interrupt condition and program line number. Nothing else happens when the statement is executed. But when the interrupt condition occurs, program control is transferred to the specified line number as soon as the current command is complete.

For example, a program might contain the statement

```
ON SRQ THEN 120
```

This statement tells the 4052 to jump to line 120 when an SRQ occurs. When it is executed, nothing obvious happens, but a link is established in the 4052 between the SRQ condition and line 120. This link remains valid until another ON SRQ statement is executed, the 4052 power is turned off, or an OFF SRQ statement is executed.

If an interrupt occurs for which no ON statement is in effect, an error message is printed on the 4052 display and program execution is terminated.

The OFF statement prevents the current program from responding to the specified interrupt condition. OFF provides a convenient means of disabling interrupt conditions set up with a previous ON statement. The syntax of the OFF statement is:

$$\text{OFF} \left\{ \begin{array}{l} \text{EOF(0)} \\ \text{SIZE} \\ \text{EOI} \\ \text{SRQ} \end{array} \right\}$$

EOF interrupts. The EOF interrupt occurs when the internal magnetic tape unit reaches the logical end of a file. When the EOF condition is specified in an ON statement, the logical unit number (0) must specified along with the keyword EOF. For example, the statement

```
ON EOF(0) THEN 500
```

transfers program control to line 500 when the logical end of the current magnetic tape file is reached. This facility can be used to read data from a file of unknown length or to find the end of an existing file to add data to it.

EOI interrupts. An EOI interrupt is generated whenever an external device asserts the EOI line on the GPIB. In most cases, EOI is used to indicate the last byte of a message. The talker asserts this line with the last byte in the message.

This interrupt is normally only used when a transfer is set up between two devices without the

4052's involvement. During the transfer, the 4052 relinquishes control of the bus to the talker. Since EOI is asserted with the last byte in the message, it can be used to tell the 4052 when the external transfer is complete and it can take control of the bus again. The use of the EOI interrupt is described more fully in **Transfers among GPIB instruments**.

It is not necessary to execute an ON EOI statement when the 4052 is involved (talking or listening) in the data transfer.

SIZE interrupts. A SIZE interrupt is generated when a numeric overflow occurs in the current program. In general, SIZE errors are caused by computations that result in out-of-range numbers. The numeric range of the 4052 is -1.0E+308 to 1.0E+308.

SRQ interrupts. Part of the GPIB system controller's responsibilty is to handle SRQ interrupts from instruments on the bus. An instrument may assert SRQ for any number of reasons, including power-up, command errors, internal errors, operation complete, etc. In any case, the instrument expects the controller to respond by reading its status byte. Reading the status byte accomplishes two things: it tells the controller why the instrument asserted SRQ, and it clears the interrupt.

The status byte contains information about the instrument's internal operations, error conditions, or other information that is important to the controller. The IEEE 488 standard reserves bit 7 of the status byte to indicate whether an instrument is asserting SRQ or not. If the instrument is asserting SRQ, bit 7 is set; if not bit 7 is cleared.

Since the SRQ line is OR'ed among all instruments on the bus, any one asserting SRQ causes the line to be asserted. As a result, the 4052 can't tell which instrument is asserting SRQ. It must read the status bytes of each instrument, looking for one with bit 7 (SRQ) set. This process is called a serial poll and it is implemented with the POLL command in 4050 BASIC. A typical POLL statement is shown below.

```
POLL X,Y;7;1,1;2
```

Two numeric variables are specified in the POLL statement (X and Y in our example), followed by a list of GPIB addresses. Each address is delimited by semicolons. If a secondary address is included it is separated from the primary address by a comma.

The 4052 begins by reading the status byte from the first instrument in the list (address 7), then the second (primary address 1, secondary address 1), then the third (address 2), and so on until the device that is asserting SRQ is found (indicated by bit 7 being set in the status byte). When the device that is asserting SRQ is found, its position in the address list is returned in the first variable (Y), and its status byte is returned in the second variable (X).

Normally, the POLL statement is executed as part of an interrupt service routine called by an ON SRQ statement. The program in Fig. 4-6 illustrates a simple case of an SRQ handling routine called by an ON SRQ statement.

```
10 DIM D(3)
20 D(1)=7
30 D(2)=1
40 D(3)=2
50 ON SRQ THEN 200
60 GO TO 60
   .
   .
   .
200 POLL X,Y;7;1,1;2
210 PRINT "SRQ FROM DEVICE NUMBER ";D(X);" STATUS: ";Y
220 RETURN
```

Fig. 4-6. *A simple SRQ handling routine called by an ON SRQ statement.*

Lines 10-40 set up a three-element array that contains the device addresses. The first device is set for primary address 7, the second is set for address 1, and the third is set for address 2. Then, line 50 tells the 4052 to execute the routine starting at line 200 when an SRQ occurs on the GPIB. Line 60 forms a continuous loop that keeps the 4052 busy while it's waiting for the interrupt.

When the SRQ occurs, the 4052 jumps to line 200 and begins polling the devices. The first device found asserting SRQ causes the polling process to stop. The POLL command returns the status byte of this instrument in Y and the position of the device in the address list in X. The value returned in X selects the element of array D that contains that instrument's address and line 210 prints the address and status byte. Finally, line 220 returns control to the loop at line 60, where it stays until another SRQ occurs.

Status byte format. Tektronix instruments report two types of status bytes when polled—system status bytes and device status bytes. System status bytes define conditions that are common among all instruments that conform to the Tektronix Codes and Formats Standard. Device status bytes define conditions that may be unique to the type of instrument.

System status bytes are further divided into normal and abnormal system status. Normal conditions include power-up and operation complete. Abnormal conditions include command error and internal error. The two types of status are differentiated by bit 6 of the status byte. Normal conditions have bit 6 cleared (zero) and abnormal conditions have bit 6 set (one).

In general, the status byte contains the following information:

BIT 8 -	System status=0 Device status=1
7 -	SRQ not asserted=0 SRQ asserted=1
6 -	Normal condition=0 Abnormal condition=1
5 -	Not busy=0 Busy=1
4 -	encoded device/system status
3 -	encoded device/system status
2 -	encoded device/system status
1 -	encoded device/system status

The system status bytes and device status bytes and their meanings are defined in the instrument manuals.

Processing the status byte. Once the status byte for an instrument has been read, the 4052 may need to take some action based on what the status byte says. Status byte processing for Tektronix instruments can be broken into two major parts—processing system status bytes and processing device status bytes.

Since the system status bytes for Tektronix instruments are common, they can be processed without regard for the specific instrument that generated them. For example, a decimal 97 status byte means that an instrument has received a command that it does not understand. The meaning of this byte is common for all Tektronix instruments.

System status bytes all have a zero in bit 8, so their decimal value is 127 or less. This provides a convenient means of testing whether a status byte is a system status or device status. If the byte is a system status byte, it can be processed by a common routine for all instruments. If it is a device status byte, separate routines that handle the device-specific status bytes are called.

A simple technique for processing the system status bytes is illustrated in Fig. 4-7.

```
200 POLL X,Y;7,1
210 PRINT "SRQ FROM INSTRUMENT ";
220 IF Y>=128 THEN 500
230 IF Y<96 THEN 250
240 Y=Y-30
250 Y=Y-63
260 GOSUB Y OF 280,300,320,340,360,380,400,420,440
270 RETURN
280 PRINT "** SRQ QUERY REQUEST **"
290 RETURN
300 PRINT "** POWER-UP **"
310 RETURN
320 PRINT "** OPERATION COMPLETE **"
330 RETURN
340 PRINT "** COMMAND ERROR **"
350 RETURN
360 PRINT "** EXECUTION ERROR **"
370 RETURN
380 PRINT "** INTERNAL ERROR **"
390 RETURN
400 PRINT "** POWER FAIL **"
410 RETURN
420 PRINT "** EXECUTION WARNING **"
430 RETURN
440 PRINT "** INTERNAL WARNING **"
450 RETURN

500 REM ** DEVICE
```

Fig. 4-7. *A routine for processing system status bytes from Tektronix instruments.*

The program starts in line 200 by polling the instrument. A single instrument is assumed, but the same technique could be expanded to handle several instruments. Line 210 prints the first half of the message. The semicolon at the end of the PRINT statement inhibits the carriage return usually added to the end of the print statement so that the rest of the message can be printed on the same line.

Then, line 220 checks that the status byte is a system status. If it is greater than or equal to 128 (decimal), it is a device dependent status, and separate routines are called to process the byte. These routines could be added starting at line 500.

Line 230 separates the byte into normal condition and abnormal condition. If the status byte is less than 96, it is a normal condition system status. Line 250 subtracts 63 from the status byte to reduce the byte to a number between 1-3. If the status byte is greater than or equal to 96, line 240 subtracts 30 from it and line 250 reduces it to a number between 4-9. The resulting value is used as an index to select one line number from the list in the GOSUB statement.

Assume, for instance, that the instrument has just been powered-up. When line 200 is executed the power-up status byte (65) is returned in Y. Line 210 prints the first half of the message. Since the status byte is less than 128, the condition of line 220 is not satisfied, and the GOTO 500 is not executed. Instead, line 230 is executed. The byte is less than 96, so control is passed to line 250, where 63 is subtracted from the status byte. The result is 2 so the GOSUB statement in line 260 sends control to the second line number in the list, line 300. Line 300 prints the last half of the message—POWER UP. The complete message printed on the terminal is:

```
SRQ FROM INSTRUMENT ** POWER-UP **
```

Processing device-dependent status. Since the device-dependent status bytes are often unique to each instrument, individual routines are usually required to process the status bytes. When a status byte is determined to be device-dependent (greater than or equal to 128), individual processing routines can be called based on which instrument generated the interrupt.

The simplest method of calling the individual routines is to use a computed GOTO or GOSUB statement. The first variable returned from the POLL statement can be directly used as the index for the GOTO or GOSUB statements. Consider, for example, the program segment shown in Fig. 4-8.

Four instruments are polled in line 10. If the status byte is a system status byte, control is passed to line 400 regardless of which instrument generated the SRQ. If the status byte is a device-dependent byte, the value returned in X indicates which instrument was asserting SRQ. Line 30 uses this value to select the interrupt handling routine for that instrument. Notice that similar instruments may use the same routine, as the first and third instruments do in this example.

```
10 POLL X,Y;5;10;7;2
20 IF Y<128 THEN 400
30 GOSUB X OF 100,200,100,300
40 RETURN
  .
  .
  .
100 REM PROCESS STATUS BYTES FROM INSTRUMENTS #1 and #3
  .
  .
  .
200 REM PROCESS STATUS BYTES FROM INSTRUMENT #2
  .
  .
  .
300 REM PROCESS STATUS BYTES FROM INSTRUMENT #4
  .
  .
  .
400 REM PROCESS SYSTEM STATUS BYTES
```

Fig. 4-8. *The computed GOTO and GOSUB statements make calling individual device-dependent status processing routines simple. This routine illustrates the use of a computed GOSUB.*

Using the WAIT statement. The WAIT statement in 4050 BASIC provides a way of temporarily halting execution of a program while waiting for an interrupt to occur. An ON statement should be executed for the interrupt. When the interrupt occurs, control is transferred to the line number specified in the ON statement.

The WAIT statement can be used to synchronize instrument and controller operations. For example, many acquisition instruments generate an SRQ when an acquisition sequence completes. The WAIT statement can be used to delay reading the data from an instrument until the acquisition complete interrupt occurs. The program in Fig. 4-9 illustrates this technique used with a 492P Programmable Spectrum Analyzer.

The program sets the 492P to single sweep mode, turns on the end-of-sweep interrupt and arms the sweep in line 30. Line 40 halts execution of the program until an interrupt occurs. When it does, control is transferred to line 100. The instrument is polled and the status byte is tested. If the end-of-

```
10 ON SRQ THEN 100
20 F=0
30 PRINT @1:"SIGSWP;EOS ON;SIGSWP"
40 WAIT
50 IF F=0 THEN 40
60 PRINT @1:"CURVE?"
70 INPUT A0
.
.
.
100 POLL X,Y;1
110 IF Y<>66 THEN 130
120 F=1
130 RETURN
```

Fig. 4-9. *A sample program using WAIT to synchronize the 4052 with a 492P Programmable Spectrum Analyzer.*

sweep status (66) is returned, a flag variable, F, is set to 1. Any other status byte causes the flag to remain zero.

After the interrupt is serviced, control returns to line 50. If the flag is set. indicating that the end-of-sweept interrupt occured, execution continues. Otherwise, control is returned to the WAIT statement. Lines 60 and 70 send the CURVE? query and read the data into array A0.

Polling instruments that are not asserting SRQ. Occasionally, the status of an instrument must be read even when it is not asserting SRQ. For example, a program may need to check an instrument's status before sending a command. The POLL command looks for instruments asserting SRQ by checking the SRQ bit (7) of the status byte. If no instruments are found asserting SRQ (bit 7 set), the POLL command returns a zero for the status byte.

Another way to read the status byte is to use WBYTE and RBYTE to send the necessary interface messages and retrieve the status byte from the instrument. This technique allows you to read the status byte whether the instrument is asserting SRQ or not. It can also be useful if you are dealing with an instrument that does not set bit 7 of its status byte to indicate SRQ. The program segment shown in Fig. 4-10a illustrates this process.

Line 10 unaddresses any talkers by sending the UNTalk message (95). Then, it sends the Serial Poll Enable message (24), followed by the talk address of an instrument set for address 3 (67). The serial poll enable command causes the instrument to put its status byte on the bus when addressed to talk. Line 20 reads the status byte and stores it in S0. Finally, line 30 unaddresses the instrument and sends the Serial Poll Disable message (25).

```
10 WBYTE @95,24,67:
20 RBYTE S0
30 WBYTE @95,25:
```

a. *Three simple routines read the status from a single instrument.*

```
10 WBYTE @95,24:
20 FOR I=1 TO 3
30 WBYTE @64+I:
40 RBYTE S0(I)
50 WBYTE @95:
60 NEXT I
70 WBYTE @25:
```

b. *Adding a few statements to the routine in part* **a** *allows you to poll several instruments.*

Fig. 4-10. *Two routines for polling instruments that are not asserting SRQ.*

This program can also be enhanced to read several status bytes by adding looping statements shown in Fig. 4-10b.

The basic process is the same as described for the program in part **a** of the figure, except that on each pass through the loop the next instrument is addressed to talk and its status byte is read. The talk address for each new instrument acts as an implied UNTalk command for the preceding instrument. When all instruments are polled, the UNTalk command is sent to explicitly unaddress the last instrument. Finally, the serial poll disable message is sent.

Using GPIB Peripherals

A variety of GPIB-interfaced peripheral devices are also available for program and data storage, graphic and alphanumeric output, and data logging. Tektronix manufactures several GPIB peripheral devices particularly designed for compabilility with the 4052. When these devices are used, the mechanics of addressing and controlling the peripherals are handled automatically with high-level 4050 BASIC statements.

The 4052's powerful I/O system allows you to address Tektronix GPIB peripherals with the same high-level BASIC statments used to address internal devices, such as the graphic display or tape drive. All you have to do is specify the peripheral device number of the external peripheral device in the I/O statement. For example, to draw a line on a GPIB-interfaced 4662 plotter, use the statement:

```
DRAW @1:X,Y
```

This statement draws a line on the plotter from the current pen position to the coordinates specified in X and Y. The only difference between this statement and the equivalent statement for for 4052's internal graphic display is the peripheral device number.

This section takes a brief look at three Tektronix GPIB peripherals and their operation with the 4052. More complete information on their operation is contained in their Operators manuals.

4907 Flexible Disk File Manager. One such peripheral device is the Tektronix 4907 File Manager. The 4907 provides fast random access flexible disk storage for 4050-series Graphic Computing Systems. Up to 630K bytes of program and data storage is available on each disk with up to three disks per system. The 4907 comes with a special ROM pack that adds several new commands to the 4052's vocabulary. These commands provide the following functions:

- File naming
- File security with passwords
- Automatic increase in file space when necessary
- File copying
- Multiple file access
- Recording time and date of all file activities
- File renaming
- Five file storage levels
- Fast random access files

Like the other ROM packs, the 4907 operating system software occupies no RAM memory, so space is left free for user programs and data. The easy-to-learn plain English commands also make programming the 4907 a simple task. For example, to create a new file, use the CREATE command. A typical CREATE command is shown below.

```
CREATE "FILE.JNK";100,128
```

This command creates a file named FILE.JNK in the current library with 100 records of 128 bytes/record. (For more information on the CREATE command, refer to the 4907 Operator's Manual.)

The 4907 also provides five levels of file storage through the use of "libraries." A library contains the names of other files or libraries, and it is used to group files. Figure 4-11 shows the simplest form of file structure on a 4907 disk. This disk contains a single level of files. No libraries are used.

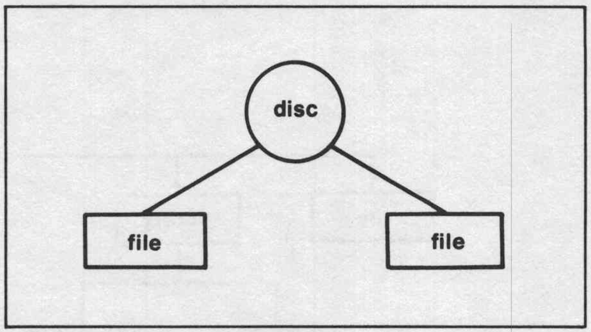


Fig. 4-11. *The simplest file storage structure on the 4907 contains only a single level with no libraries.*

When a library is added, files are placed on the next lower level, as shown in Fig. 4-12. This structure can be extended for up to five levels as shown in Fig. 4-13.

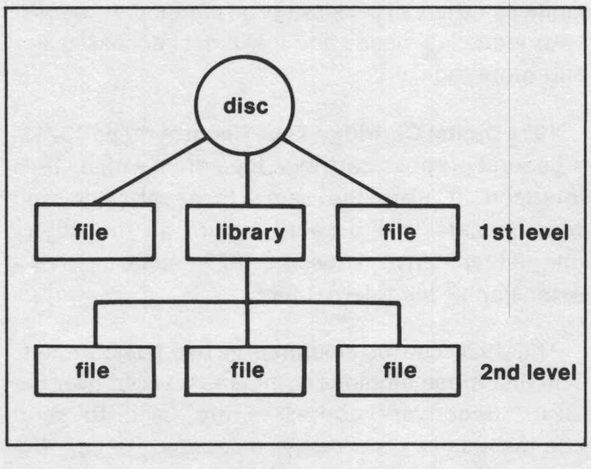


Fig. 4-12. *Libraries are used to group files on the subsequent levels. Files and libraries may be mixed on any level except the last.*

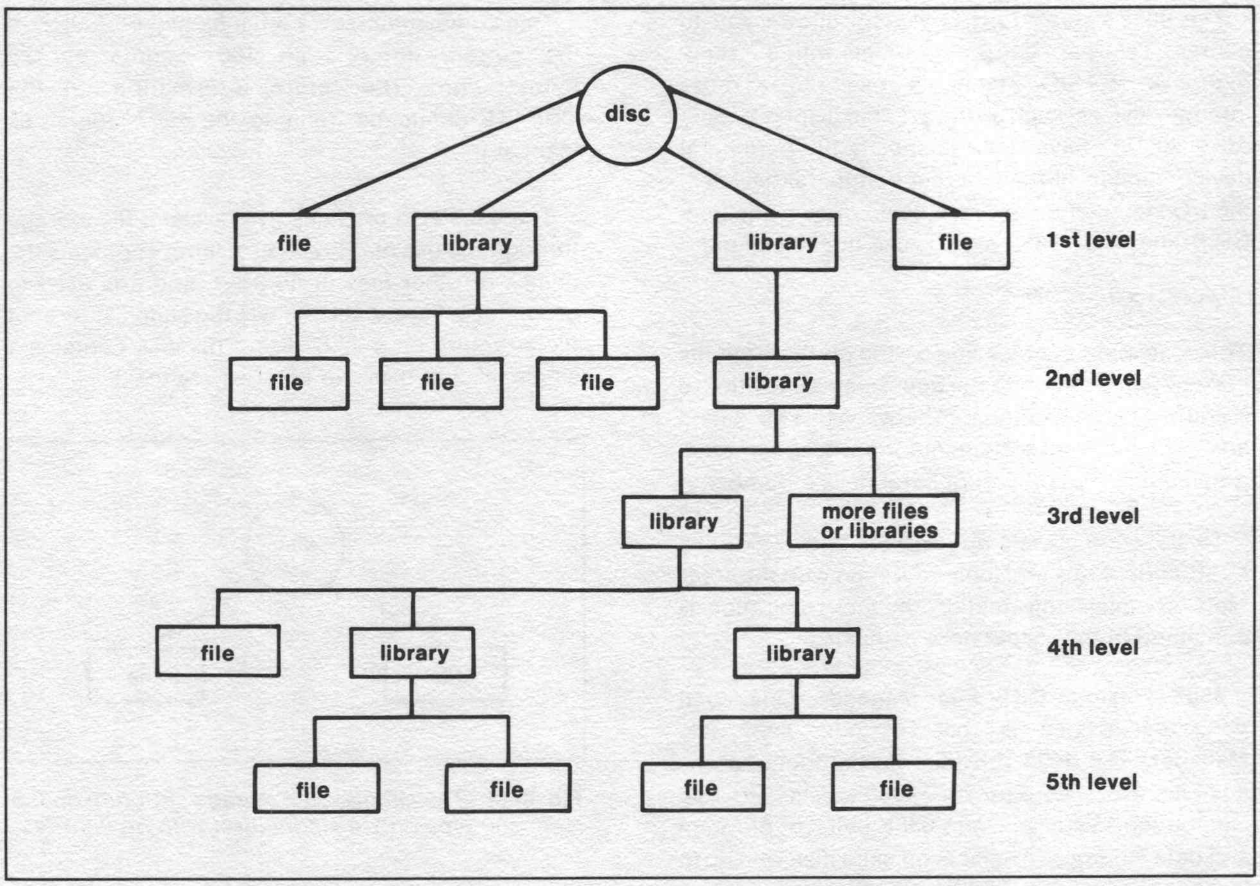


Fig. 4-13. *The 4907 file structure can be expanded to five levels. Files and libraries can be mixed on any level except the fifth.*

This five-level storage structure allows an almost limitless variety of file storage arrangements to meet your individual needs and make data access easier and more logical.

4924 Digital Cartridge Tape Recorder. The 4924 is a general purpose cartridge tape drive with a GPIB interface. It uses the same tape cartridge and records data in the same format as the 4052's internal tape drive. Thus, the 4924 can be used as a extension of the internal tape.

The 4924 can be operated in two basic modes. The first mode employs commands issued over the GPIB. Secondary addresses are used to send commands as previously discussed under **I/O Addressing in 4050 BASIC**. Since the command codes are the same for the 4924 and the internal 4052 tape drive, the interface is simple. For example, to kill a file on an external 4924, use the KILL command:

```
KILL @2:5
```

This command kills file number 5 on a 4924 set for primary address 2. The 4052 automatically sends the secondary address that tells the 4924 to execute a KILL operation.

In addition, commands can be sent to the 4924 using device-dependent ASCII messages in place of the secondary address. For example, to execute the KILL command described above, the controller sends the command primary address (separate command and data primary addresses are used in this mode). Then, with attention unasserted, the letter "K" is sent followed by a delimiter (space, comma, or semicolon), the file number, and a carriage return. Finally, the unlisten message is sent with attention asserted.

In the second mode, the 4924 operates manually from the front panel. The front panel buttons are used to perform basic tape operations such as advance forward or reverse, talk, or listen. This mode is most useful when the 4924 is operated as a data logging device without a controller.

4662 Interactive Digital Plotter. The Tektronix 4662 is an interactive plotter with RS-232C and GPIB interfaces. The plotter can print alphanumerics and graphics on paper or other media up to 11 by 17 inches. In addition, it can perform as a graphic input device.

When interfaced via the GPIB, the 4662 accepts commands in one of two modes, like the 4924 tape drive. In the first mode, commands are sent as secondary addresses. The command codes correspond to those used in the 4052, so the plotter can be fully controlled using simple high-level BASIC statements. For example, to move the plotter pen to a specific location, execute the MOVE command:

```
MOVE @1:X,Y
```

where: X is the horizontal coordinate in Graphic Display Units; Y is the vertical coordinate in Graphic Display Units

4050 BASIC includes statements to execute relative and absolute MOVEs and DRAWs, to generate axes, rotate and scale alphanumerics, and to window and scale graphic data.

The 4662 also implements another command mode for interfacing with other controllers. In this mode, device-dependent messages are used to send plotter commands. Normally, this mode is not used with 4050-series controllers, since the secondary address scheme is implemented automatically in 4050 BASIC. The 4662 Operator's manual contains complete information on using this mode.

Section 5—Processing and Displaying Data

Most GPIB systems make some type of measurement as part of their job, whether it be a simple voltage measurement or a complete waveform acquisition. The first step in making the measurement is to acquire the signal and convert it to a digital format. That is the job of the acquisition instrument (i.e. waveform digitizer, digital voltmeter, etc). The output of this instrument is then sent to the controller over the GPIB (Fig. 5-1).

Once data is acquired and transferred to the 4052, some processing is often required to derive the desired information. It's important to remember that the 4052 is processing a digital representation of the input signal—a string of numbers—not the signal itself. The numbers usually represent vertical signal amplitudes at discrete sample points along the signal. The position of each number in the series represents its horizontal time location on the signal (Fig. 5-1).

When the data is processed, the 4052 manipulates only the numbers stored in it's memory. Almost without exception, the processing is done on an element-by-element basis, starting with the first element in the array and progressing to the last one.

For example, let's say you have acquired a waveform, and it has been transferred into an array in the 4052. Now, maybe you want to add a constant to it. The data might represent a voltage waveform to which you want to add a four-volt bias. The 4050 BASIC statement is:

```
A=A+4
```

When this statement is executed, the 4052 adds four to the first element in array A. Then it adds four to the second element, and the third, and so on until all the elements have been processed.

This same element-by-element process is also used in subtracting a constant from an array, multiplying an array by a constant, or most any other arithmetic operation. It is also used when two arrays are processed in a statement, such as multiplying two arrays. For example, the statement

```
A=B*C
```

causes the 4052 to multiply each element of array B by the corresponding element in array C, and store the result in array A.

This all seems so simple. And it is—if you avoid the more common pitfalls by keeping the following DOs and DON'Ts in mind:

DON'T attempt to combine (add, subtract, multiply, or divide) arrays of different lengths since the element-by-element processing won't complete. If you attempt such an operation, the 4052 will remind you by printing an error message.

DON'T attempt to combine data arrays acquired at different sampling intervals. For example, don't add a waveform array that was acquired at 5 microsecond sampling interval with a waveform sampled at 15 microseconds/sample. The time scaling may lead to erroneous or confusing results.

DO be cautious of dividing by zero or very small numbers (such as occur at zero crossings on repetitive waveforms) since this can lead to SIZE errors or erroneous results.

DO keep in mind the limits of the 4052's calculation accuracy. All math operations are computed to 14 digits of accuracy, so for all but the most lengthy calculations, the round-off error is insignificant. But, it's important to remember that round-off error can accumulate in lengthy calculations until it becomes significant.

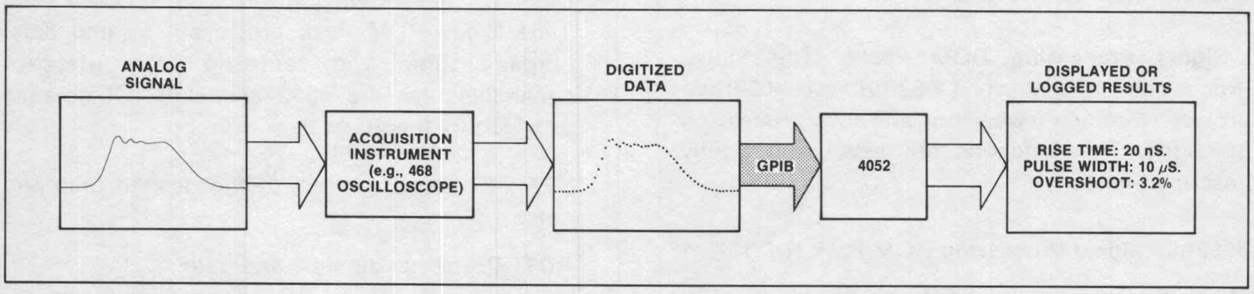


Fig. 5-1. *Automated measurement begins with the acquisition instrument. It sends digitized data to the 4052. The data may either be logged or processed and the results logged or displayed.*

Using the ROM pack functions. Most signal processing needs go beyond simple mathematical combinations of constants and waveforms. Often, special signal processing or array operations are required, such as finding the maximum or minimum of an array, integrating, differentiating, or other complex operations. To help speed this type of processing, many of the common signal and array processing functions are provided in a series of ROM (Read-Only Memory) packs that plug into the slots in the rear of the 4052.

The ROM pack commands execute much faster than equivalent programs written in BASIC. Some commands execute as much as ten times faster! Also, the ROM pack commands do not occupy any user memory space.

ROM pack functions are executed with a simple CALL statement. The syntax of the statement is

```
CALL "routine name",parameter[,parameter...]
```

For example, to find the maximum value in an array, use the MAX function. The function can be called with this statement:

```
CALL "MAX",A,V,I
```

Where A is the source array that is searched. The maximum value of the array is returned in V, and the location (subscript) in the array of the maximum value is returned in I.

The routine name may also be stored in a string variable and substituted for the name in the CALL statement, as illustrated below.

```
A$="MAX"
  .
  .
  .
CALL A$,A,V,I
```

Signal Processing ROM Packs. The Signal Processing ROM Packs (4052R07 and 4052R08) provide 15 common waveform and array processing functions. The functions are listed and briefly described below.

4052R07 Signal Processing ROM Pack No. 1

MAX—finds the maximum of an array.

MIN—finds the minimum of an array.

CROSS—finds the location of a specified crossing level within an array.

DIF2—performs a two-point differentiation of an array.

DIF3—performs a three-point differentiation of an array.

INT—integrates an array.

DISP—displays a graph of an array in raw form (without axes).

4052R08 Signal Processing ROM Pack No. 2

FFT—computes the fast Fourier Transform of a one-dimensional array.

IFT—computes the inverse Fourier transform of a one-dimensional array.

CONV—convolves two one-dimensional arrays.

CORR—correlates two one-dimensional arrays.

POLAR—converts an array of complex data from rectangular form (real and imaginaries) to polar form (magnitude and phase).

TAPER—multiplies an array by a cosine window of program-selectable weights.

UNLEAV—sorts an array of interleaved FFT data into two arrays, one containing real and one containing imaginary components.

INLEAV—interleaves the real and imaginary data from two input arrays into a third array whose format is acceptable to the IFT command.

Real-Time Clock ROM Pack. In many real-time applications, such as automated testing, the time when an event occurs can be as important as the event or measurement itself. The 4052R09 Real Time Clock ROM Pack provides time and date, elapsed time, and vectored time interrupt capabilities for the 4052 and 4054. Commands provided by this ROM pack are:

SETIME—sets the clock to the desired time and date.

RDTIME—reads the time and date.

STARTW—resets and starts the stopwatch incrementing in 0.1 second steps.

STOPIT—reads the elaspsed time from the stopwatch.

ONTIME—sets the programmable interrupt delay. When the delay expires, control is transferred to line 84 of the user program.

Graphing data. The old adage "a picture is worth a thousand words" may be a bit hackneyed but it is still true—particularly in the realm of science and engineering. An important relationship involving two or more variables may be difficult, if not impossible, to understand when presented as a column of numbers. Yet, by means of a graph or chart, the same relationship can often be recognized at a glance.

In addition, ideas or information that are difficult to convey in words can often be easily conveyed in pictures. For example, describing a test set-up in words can be difficult, especially when low-skill operators are involved. But a picture of the test set-up provides the same, if not more, information for the operator in simple terms that are easily understood.

The 4052's powerful, high-resolution graphics capability and extended BASIC language make generating and displaying graphics a simple task. Statements are included in 4050 BASIC to scale data, map it into a defined window, move the window anywhere on the screen, and perform a variety of graphic functions within that window.

Data can be defined in any units appropriate to the application. For example, a program that tests frequency response of a system might use decibels for vertical axis units and frequency for the horizontal axis units. Once these units are declared and their limits defined, the 4052 automatically maps the user's data units into the graphic display.

Sample program. The program in Fig. 5-2 illustrates the use of several signal processing ROM pack functions as well as some simple graphics functions. It performs some basic pulse analysis on a waveform stored in array A. The program assumes a simple pulse is stored in array A and that its horizontal scale factor is stored in H.

The output of the program is a graph (without axes) of the input waveform and a pulse parameter summary printed below the graph.

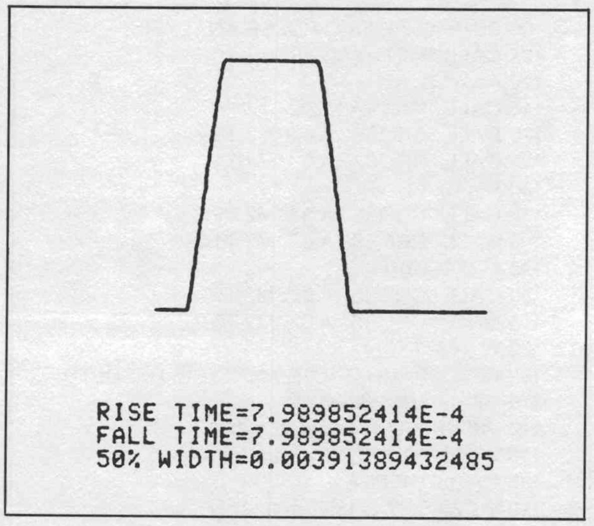
```
100 REM 4052 BASIC PULSE ANALYSIS
105 CALL "MIN",A,M1,I1
110 A=A-M1
115 CALL "MAX",A,M2,I2
120 CALL "CROSS",A,0.9*M2,T9
125 CALL "CROSS",A,0.1*M2,T1
130 R=(T9-T1)*H
135 CALL "CROSS",A,0.5*M2,T9,2
140 CALL "CROSS",A,0.5*M2,T1,2
145 F=(T1-T9)*H
150 CALL "CROSS",A,0.5*M2,T5
155 CALL "CROSS",A,0.5*M2,T6,2
160 W=(T6-T5)*H
165 REM GRAPH PULSE ANALYSIS RESULTS
170 CALL "MIN",A,M1,I1
175 VIEWPORT 6.4,44.9,52.1,82.6
180 WINDOW 1,512,M1,M2
185 CALL "DISP",A
190 VIEWPORT 0,130,0,100
195 WINDOW 0,130,0,100
200 MOVE 0,38.5
205 PRINT "RISE TIME=";R
210 PRINT "FALL TIME=";F
215 PRINT "50% WIDTH=";W
220 END
```

Fig. 5-2. *A sample program to compute some basic pulse parameters on a waveform stored in array A. The routine also graphs the waveform without axes.*

The program begins by setting the base of the pulse equal to zero to simplify processing. Line 110 finds the minimum value and subtracts this value from the entire waveform, to set the base of the pulse to zero. Line 115 returns the maximum value of the pulse in M2 and the array subscript of the maximum in I2. Then, lines 120 and 125 use this maximum value to find the 10% and 90% amplitude points and return the location of these points in T9 and T1, respectively. The difference between these points, multiplied by the horizontal scale factor (H), is the rise time for the pulse.

Line 150 finds the first 50% point and line 155 finds the second 50% point. The difference between these points multiplied by the horizontal scale factor is the 50% pulse width.

Line 170 finds the minimum value of the zero-referenced pulse. Then, line 175 reduces the size of the viewport to leave room for the pulse parameter information to be printed below. The WINDOW statement sets the limits of the data to be graphed. Line 185 displays the waveform in the current WINDOW and VIEWPORT.



```
RISE TIME=7.989852414E-4
FALL TIME=7.989852414E-4
50% WIDTH=0.00391389432485
```

Fig. 5-3. *Sample output from the pulse analysis program.*

Finally, lines 190-215 reset the VIEWPORT and WINDOW and print the pulse parameters below the graph. Fig. 5-3 shows a sample output from the program.

Section 6—Estimating GPIB System Performance

One of the most frequently asked questions about any GPIB system is "how fast will it go?" This question is often critical to the design and implementation of a system. Yet, it is often very difficult to answer. The complete performance picture is composed of a multitude of parts, many that are difficult to estimate. However, a good understanding of the basic factors that contribute to the performance of the system will help develop a good estimate of the system's overall performance.

GPIB system performance is affected by several factors. The key factors are:

- Data transfer time
- Processing time
- Data acquisition time
- Human interaction time

Data Transfer Time

A certain amount of time is required for messages to be transferred across the GPIB. This time is called the data transfer time. It includes time to transfer interface messages as well as device dependent data across the bus.

The asynchronous bus. The GPIB is an asynchronous bus. That is, data is transferred at a rate determined entirely by the instruments on the bus; there is no clock signal. Data is transferred at the rate of the slowest listener. When the talker or controller places a byte on the bus, all listeners must accept the data byte (indicated by releasing the $\overline{\text{NDAC}}$ line) before the talker can proceed to the next byte. Any listener can delay the transfer simply by holding $\overline{\text{NDAC}}$ low (asserted).

This asynchronous bus allows a variety of instruments with different speeds to work together. But, it also means that any slow device involved in a transfer slows the entire transfer down to its rate. To illustrate, consider the system shown in Fig. 6-1. The maximum data transfer rate for each device is shown in the figure.

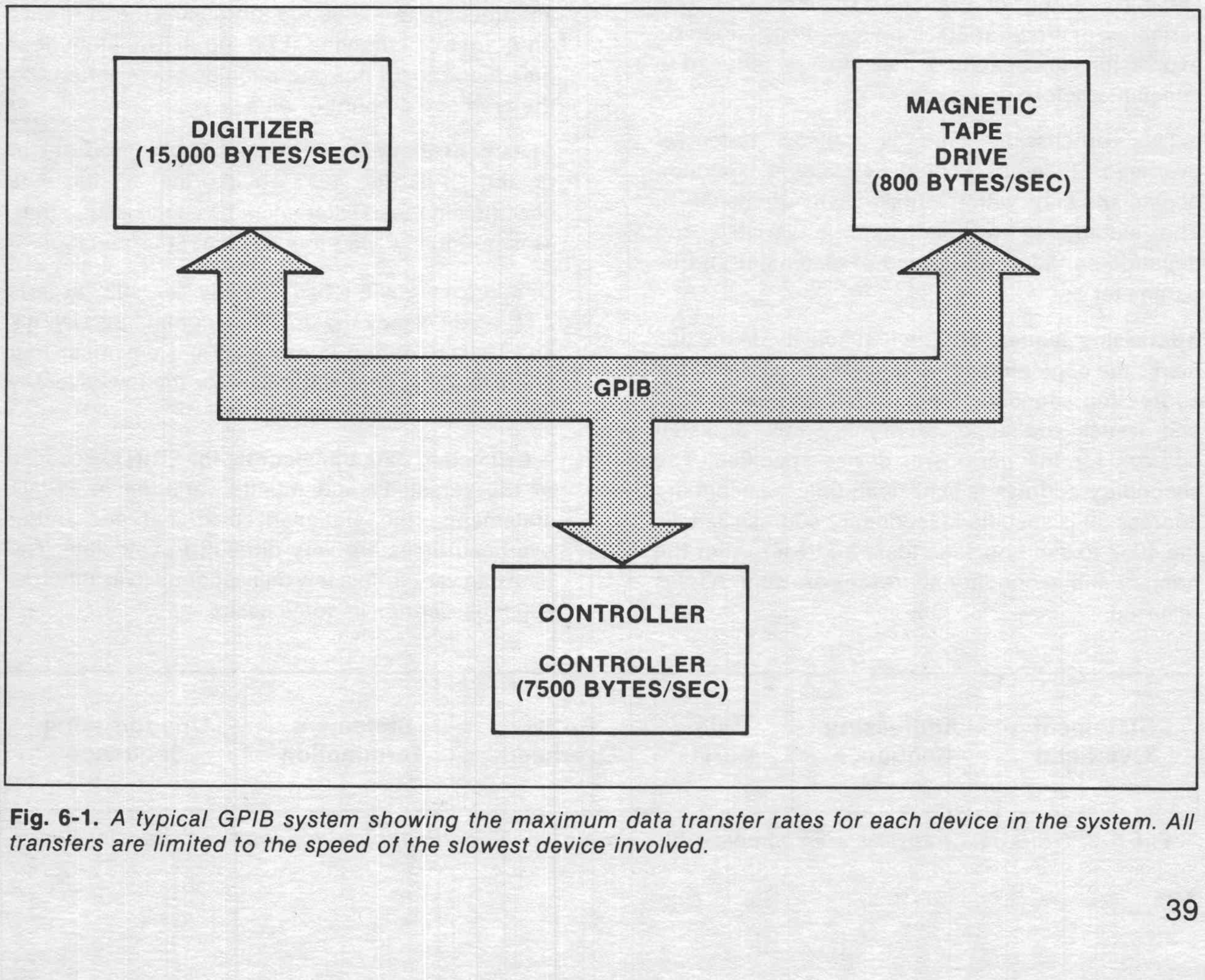


Fig. 6-1. *A typical GPIB system showing the maximum data transfer rates for each device in the system. All transfers are limited to the speed of the slowest device involved.*

When data is transferred between the digitizer and tape drive, the transfer is limited to 800 bytes/second by the tape drive. When data is transferred between the controller and digitizer, the transfer can proceed at up to 7500 bytes/second, since the tape drive is not involved. In no case will the full speed capability of the digitizer be realized, since the other two devices limit the speed of the transfers.

GPIB data transfer timing. When a 4052 I/O statement is used to transfer data on the GPIB, up to six events occur, though they don't all occur in every I/O statement. These events are graphically represented in Fig. 6-2.

Statement overhead—When an I/O statement is executed, the 4052 first examines the statement for content and syntax. For high-level I/O statements, such as PRINT or INPUT, the I/O address is checked to see if default values are necessary (e.g. when no secondary address is specified). The parameter list is also examined to see if string variables, string or numeric constants, numeric variables, or array variables are specified. If numeric expressions are specified, they must be reduced to constants before going on.

This processing time is called "statement overhead." It begins when the statement execution begins and ends with the first activity on the GPIB. The statement overhead time is variable and depends entirely on the type of statement and its parameter list.

Addressing sequence—The first activity on the bus marks the beginning of the second timing event: the addressing sequence. First, the 4052 asserts ATN and issues the absolute primary talk or listen address for the peripheral device specified. The secondary address is sent next, unless secondary address 32 is specified (secondary address 32 tells the 4052 to send no secondary address). After the primary and secondary addresses are sent, ATN is released.

Data burst—When the 4052 releases ATN, the actual data transfer begins. Data is transferred at the maximum rate determined by the slowest device involved in the transfer. On input operations, the 4052 receives bytes from the talker and stores them in an intermediate buffer memory (called the I/O buffer). On output operations, data is transmitted from the I/O buffer over the bus to the listener(s).

Buffer overhead—The I/O buffer can hold a maximum of 72 characters. If the data transfer involves more than 72 characters, the transfer is performed in 72-character bursts. On output, the 4052 converts the first 72 characters from internal binary format to the correct format for transmission and loads the formatted data into the I/O buffer. Then, the contents of the I/O buffer are transmitted.

On input, 72 characters are received and loaded into the I/O buffer. Then, this data is converted to the correct format for the variables specified in the statement and the data is stored.

Between each burst of input or output, a variable amount of time is required for refilling the I/O buffer on output or emptying it on input. The amount of time required for this operation depends entirely on the type and amount of data.

Statement termination—At the end of a data transfer on the GPIB, the 4052 checks that all the data specified in the I/O statement has been transferred. This period is called the "statement termination."

Unaddressing sequence—Finally, the 4052 asserts ATN again, sends the universal commands UNTalk and UNListen, and releases ATN. Remember that this operation does not occur for the low-level I/O statements, WBYTE and RBYTE.

Estimating data transfer rate for PRINT. Because of the versatility and infinite variation in PRINT statements, the statement overhead and buffer overhead times are very difficult to calculate. The time can vary from a few milliseconds to as much as 500 milliseconds in some cases.

Statement Overhead	**Addressing Sequence**	**Data Burst**	**Buffer Overhead**	**Statement Termination**	**Unaddressing Sequence**

Fig. 6-2. *GPIB data transfers are composed of six parts, each contributing to the total data transfer time.*

The addressing period for PRINT is constant, regardless of the format of the statement. Unless the secondary address is suppressed, the addressing period takes about 125 microseconds (minimum). With the secondary address suppressed, the addressing period takes about 90 microseconds (minimum).

When a buffer of characters is prepared for transmission, the 4052 can transmit them at about 19,800 bytes/second. This, of course, assumes that the listener is significantly faster, and does not affect the transfer rate. With a slower listener on the bus, the overhead time will not be affected, but the actual data transfer will occur at the listener's rate.

The unaddressing period takes about 150 microseconds, regardless of the statement format.

Consider this simple print statement:

```
PRINT @2:1;2;3;4;5;6;7;8;9
```

In this example, less than 72 characters are sent, so burst data rate of 19,800 bytes/second applies. Since the default PRINT statement format inserts a space before each number, the individual values (called "samples") each contain two bytes. Therefore, 19,800/2 or 9900 samples can be transmitted/second. If each sample was a two-digit number, the rate would be reduced to 19,800/3 or 6600 samples per second.

This same idea applies to numeric arrays transmitted in the form:

```
DIM A(10)
A=1
PRINT @2:A;
```

Here, two byte samples are transmitted as above at a rate of 9900 samples/second.

If a comma is substituted between variables or constants in the PRINT statement or the trailing semicolon is left off an array variable, the 4052 formats each element into an 18-character field by adding spaces. Thus, every element, no matter how many digits it contains, takes 18 bytes to transmit. As a result, data samples are transmitted at 19800/18 or 1100 samples/second.

Notice that simply removing the trailing semicolon from the statement above, reduces the data rate from 9900 samples/second to 1100 samples/second! It's easy to see how a minor change in a statement can drastically affect the data rate.

Estimating data transfer rate for INPUT. The statement overhead for INPUT is a function of the statement format and the number of variables in the statement. The basic statement overhead time varies from about 2-3 milliseconds plus about 300 microseconds for each variable in the statement.

The addressing period for INPUT is essentially identical to the PRINT statement addressing period—125 microseconds with the secondary address; 90 microseconds without the secondary address.

Data bursts up to 72 characters can be received at about 102,400 bytes per second (97.5 μs per byte). Again, this assumes that the talker can send data at least this fast.

The buffer overhead period depends on the number of characters in the buffer and the type of variable they are destined for. For numeric variables, each variable requires about 1.4 milliseconds for a single digit plus about 120 microseconds per additional digit. So, to convert a single five-digit variable requires about 1.8 milliseconds (1.4 ms + 4 x 120 μs).

For string variables, the first character takes about 600 microseconds. Each additional character requires about 40 microseconds.

At the end of a transmission, the buffer is emptied whether it is full or not, and the data is assigned to variables. A final check is also made to be sure that all target variables in the statement have been assigned values. This final check is called the statement termination. It may take more or less time than the buffer overhead, depending on the number of characters left in the buffer and the type of variables they are assigned to.

Finally, it takes the 4052 about 150 microseconds to assert ATN, send the UNTalk and UNListen commands, and release ATN.

For example:

```
10 DIM X(1000)
20 INPUT @4,32:X
```

Device 4 sends 1000 data samples with five digits in each sample. A comma delimits each sample. A typical data stream might look like this:

36524,37428,39266,39694...<996 more samples>

The time to execute the INPUT statement in line 20 is estimated as shown below.

Statement overhead 2.5 ms + 300 µs ≅	2.8 ms
Addressing period ≅	90 µs
First 83 data bursts 72 chars. x 97.5 µs/char ≅	582 ms
Last data burst 24 chars. x 97.5 µs/char ≅	2.34 ms
Buffer overhead:	
2.0 ms/sample x 1000 samples ≅	2.0 sec
Statement termination ≅	1.2 ms
Unaddressing period ≅	150 µs
Total data transfer time ≅	2.59 seconds
Effective data sample transfer rate ≅	386.1 samples/sec.

Estimating data transfer rate for WRITE. The WRITE statement transfers data over the GPIB in 4052 internal binary format. Each data item is preceded by a two-byte header which identifies the data item type (number or string) and the length of the item (in bytes). The length of a numeric data item is always eight bytes plus the header. The length of a character string is one byte per character plus the header.

Since data is sent in 4052 internal binary format, WRITE is of little practical value for sending data to a device other than a storage device (such as the 4907 File Manager). Data written to a storage device can later be loaded directly back into the 4052.

The addressing period for WRITE is the same as any other high-level I/O statement. It takes about 150 microseconds to send a primary and secondary address. The primary address alone takes about 90 microseconds to send.

The data rate changes during different parts of the data burst, so an absolute number can be misleading. The shortest transfer time for a 10-byte numeric data item is about 800 microseconds, which translates to about 12,500 bytes/second.

For string data, the main part of the string is sent at about 3100 bytes/second. Long strings and the two-byte header are transmitted slower.

The unaddressing period for WRITE, like all other high-level I/O statements, takes about 150 microseconds.

Estimating data transfer rate for READ. The READ statement is generally used to bring data in 4052 internal binary format back from a peripheral storage device. It is usually not practical to receive data from an instrument with READ, since most instruments cannot transmit data in the required format.

When receiving numeric data, each sample consists of two bytes of header plus eight bytes of floating point binary data. The maximum sustained data rate for numeric data is 875 samples/second.

The maximum rate for string data is determined by the number of characters in the string. The total time to transfer a string can be estimated from the following equation:

Transfer time = 640 µs + ((No. of chars.–1) x 50)µs

Estimating data transfer rate for WBYTE. The WBYTE statement is normally used in situations where PRINT or WRITE can't be used. An example is transmitting binary data to an instrument (not 4052 internal binary format), or setting up a transfer among two or more devices. It is also used to transmit interface messages.

The statement overhead period for WBYTE is shorter than for PRINT and WRITE because part of the overhead is performed during the statement. The overhead generally falls in the range of 1.6 to 2.4 milliseconds. Following are general rules for estimating the statement overhead.

1. The statement overhead is reduced if a decimal value is assigned to a numeric variable, then specified as a variable in the WBYTE statement.

2. Specifying one peripheral address results in a statement overhead of about 1.6 milliseconds. Each additional address adds approximately 500 microseconds to the statement overhead. If the addresses are specified as variables, each additional address adds approximately 200 microseconds to the overhead.

3. Each numeric data byte specified after the colon adds approximately 520 microseconds to the statement overhead. If the data bytes are specified as variables, each variable adds about 160 microseconds to the statement overhead.

Table 6-1 shows several example WBYTE statements and the estimated overhead times for each.

TABLE 6-1

EXAMPLES OF ESTIMATING WBYTE OVERHEAD PERIOD

Statement	Set-up Period
WBYTE @66:	1.6 ms
WBYTE @66,67:	2.1 ms
WBYTE @P:	1.5 ms
WBYTE @P,Q:	1.7 ms
WBYTE @67:100	2.1 ms
WBYTE @67:100,100	2.6 ms
WBYTE @67:A	1.8 ms
WBYTE @67:A,B	2.0 ms

Remember that WBYTE does not perform the automatic addressing and unaddressing functions that PRINT and WRITE do. Therefore, the addressing and unaddressing periods do not apply to WBYTE.

The conversion of each value specified in a WBYTE statement to binary occurs just before the value is transmitted. As a result, the data rate varies for each variable. The only practical method for accurately estimating the data rate is by actual testing. To get a rough idea of how fast WBYTE is, consider this example:

```
10 DIM A(50)
20 A=2
30 WBYTE @34:A
```

The WBYTE statement in line 30 transfers data at 2975 bytes/second.

Estimating data transfer rate for RBYTE. The RBYTE statement is used to receive data bytes one at a time from the GPIB. It is normally used to receive binary data (not in internal 4052 binary format) from an instrument. Since INPUT receives ASCII data and READ requires 4052 internal binary format, RBYTE is the only choice for binary data sent by some instruments.

The statement overhead period for RBYTE can be estimated as follows:

Statement overhead ≅ 1.2 ms + no. of variables x .24 ms

EXAMPLE: RBYTE A,B,C

Statement overhead ≅ 1.2 ms + 3 x .24 ms = 1.92 ms

If only one target variable is specified in an RBYTE statement, the byte can be captured in 40 microseconds. If several variables are specified, bytes can be received at about 1405 bytes/second (maximum). When an array variable is specified, the bytes can be received at 1689 bytes/second. No buffer overhead period is required, so these data rates can be sustained for any number of bytes.

Since no unaddressing occurs at the end of the RBYTE statement, the execution is complete as soon as the last byte is received.

Interface message traffic. GPIB message traffic also includes interface messages. These messages include the primary and secondary addresses used to set up device-dependent transfers, as well as other messages that implement a serial poll, device clear, or other functions. Interface messages are always sent with ATN asserted, and all devices on the bus must handshake the message, whether they are addressed or not. Thus, the slowest device on the bus regulates the transfer rate.

Interface messages consist of addresses, universal commands, and addressed commands. The addresses are automatically implemented in high-level I/O statements like PRINT and INPUT. They may also be sent with WBYTE. The most common interface commands are implemented in high-level statements such as POLL. (POLL implements Serial Poll Enable and Serial Poll Disable). The remainder of the commands must be sent with WBYTE. Estimating the transfer rate of these bytes was discussed under **Estimating the transfer rate for WBYTE**.

The serial poll process implemented with a POLL statement is probably the most common interface message traffic, and provides a good example of estimating interface message transfer time. Figure 6-3 illustrates the process of executing the following POLL statement:

```
POLL X,Y;2;7,4;3
```

Assume, for example, that the second device in the POLL list is a 7A16P Programmable plug-in installed in a 7612D Digitizer, and that the 7A16P is requesting service. The 7A16P is addressed with the mainframe primary address (3), and the mainframe secondary address plus one (4). Referring to the

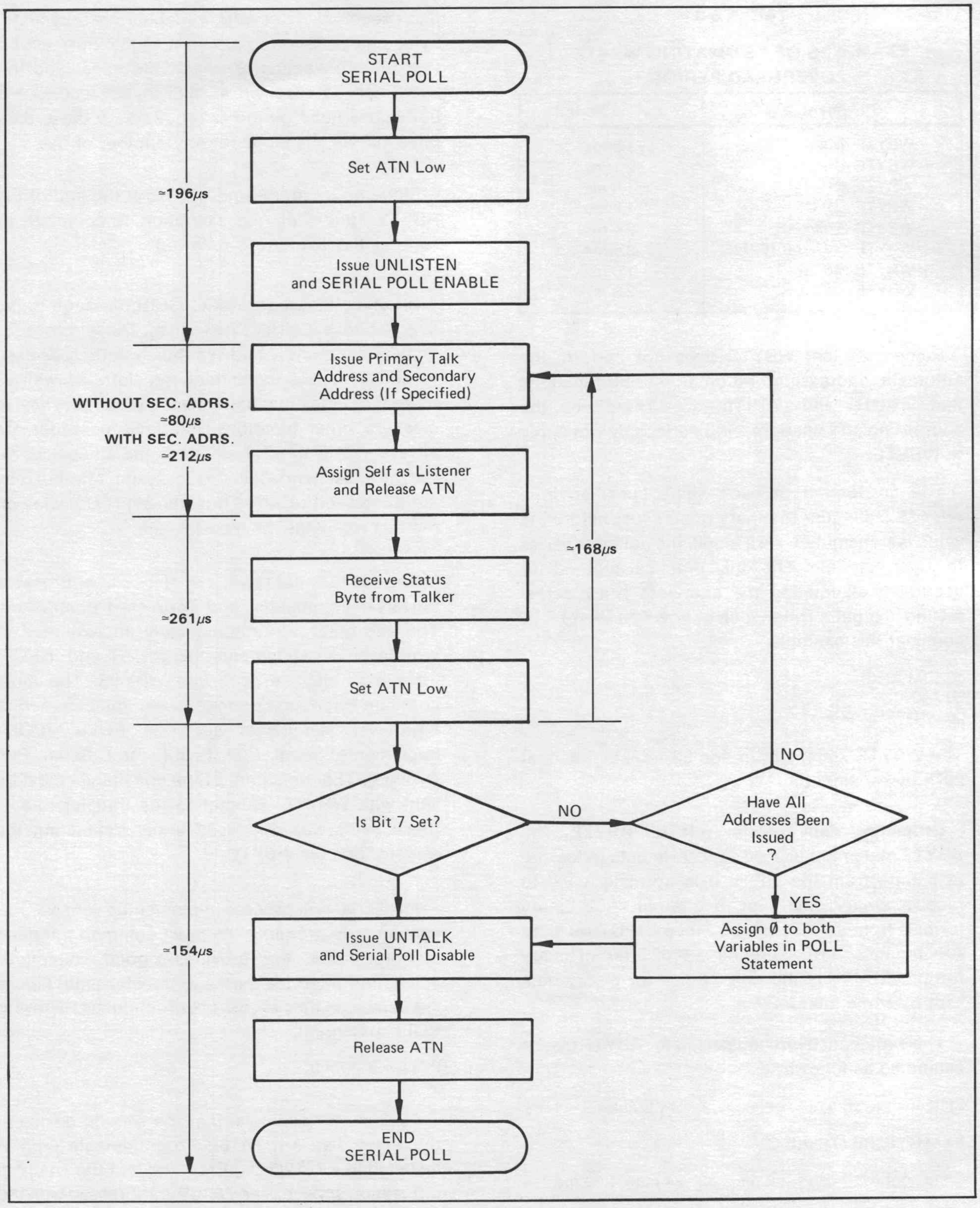

START SERIAL POLL
Set ATN Low
Issue UNLISTEN and SERIAL POLL ENABLE
Issue Primary Talk Address and Secondary Address (If Specified)
Assign Self as Listener and Release ATN
Receive Status Byte from Talker
Set ATN Low
Is Bit 7 Set?
NO
Have All Addresses Been Issued ?
NO
YES
Assign Ø to both Variables in POLL Statement
Issue UNTALK and Serial Poll Disable
Release ATN
END SERIAL POLL
≈196μs
WITHOUT SEC. ADRS. ≈80μs
WITH SEC. ADRS. ≈212μs
≈261μs
≈154μs
≈168μs

Fig. 6-3. *A flow chart of the POLL process. Approximate timing values are shown as an aid to estimating the execution time.*

timing values given in Fig. 6-3, the time required to execute the POLL statement can be estimated as follows:

Statement overhead and bus initialization ≅	196 μs
Address 1st instrument to talk (without sec. addrs.) ≅	80 μs
Get status byte from 1st instrument ≅	261 μs
Test bit 7 of status byte and loop ≅	168 μs
Address 2nd instrument to talk (with sec. addrs.) ≅	212 μs
Get status byte from 2nd instrument ≅	261 μs
Untalk instrument and disable serial poll ≅	385 μs
Total time to execute serial poll ≅	1.563 ms

Processing Time

Another significant factor in GPIB system performance is processing time—the time required by the controller and instruments to execute commands and process data. It's important to remember that in a GPIB system at least two programs are running. One program is running in the 4052, and another is running in each of the programmable instruments in the system. Each of these programs take time to execute, and that time must be accounted for in making an estimate of the system's performance.

Controller processing. In most systems, the controller's job goes far beyond simply setting up bus transfers, and collecting data. A significant amount of processing may be required in some applications to extract the desired results from the raw acquired data.

The controller processing time can be broken into two major parts:

I/O Processing
Data processing

The first of these tasks, I/O processing, involves the transfer of data over the GPIB. Estimating the data transfer time is discussed in detail in the previous section.

The data processing task is as varied as are the applications for GPIB systems. The time required to perform the processing depends on the length and complexity of the program, the speed of the controller, and the number of tasks it is assigned (e.g. how many instruments are on the bus that may interrupt the program, etc). Since these factors are very difficult to quantify, the most practical method of measuring this performance is usually actual testing. A number of programming hints are given in the next section to help improve the performance of 4052 GPIB systems.

Instrument processing. Most programmable GPIB instruments have a microprocessor system inside that controls the GPIB and internal instrument functions. This microprocessor has three major tasks, though not all of these tasks are implemented in every instrument.

GPIB control and message processing
Instrument control
Data handling

When an instrument receives a message from the GPIB, the internal microprocessor goes to work decoding the message, checking it for errors, and taking the appropriate action. This processing takes time. However, the amount of time is often not specified in instrument manuals. As a result, the only practical way to estimate the time is by direct measurement.

The second task, instrument control, involves processing accepting and processing front-panel input and monitoring and controlling internal instrument functions.

When the instrument is set-up and the measurement taken, their may be some local processing required to get the data ready for transmission to the controller. For example, code conversions may have to be performed. In some instruments, local processing such as signal averaging or interpolation may be performed. Again, this processing takes time. If the instrument is capable of sending raw data and processed data, comparing the time required to do each task provides a reasonable estimate of the internal processing time.

Data Acquisition

The third major component in system performance is the time required to actually acquire data. In some instruments this may be an insignificant amount of time. In others, such as waveform digitizers, data acquisition can be the most time consuming operation in the system. At least two factors must be considered in data acquisition—triggering and digitizing time. And, in some applications a third component must be considered—signal averaging.

Trigger delay. Triggering is the process of starting the data acquisition at a desired point. For example, to acquire a single-shot pulse, a waveform digitizer might be set to trigger when its input rises to a certain level. Often, the system controller sets an instrument up to acquire data, but the acquisition cannot begin until a trigger occurs (Fig. 6-4).

If the trigger conditions are not satisfied, the acquisition never completes, and the whole system waits. The lesson here is: make sure the instrument(s) get the necessary trigger to begin their acquisition, and account for the time between when the controller sets up the acquisition and when the trigger occurs. Also, remember that a trigger that occurs before the controller finishes setting up the instrument is of no value.

Some instruments, like the 7612D Programmable Digitizer, have a pre-trigger and/or post-trigger feature that allows the digitizer to start acquisition before or after the trigger. In this mode, the 7612D acquires pre-trigger data before it becomes triggerable. While it is acquiring this data, triggers are ignored. This pre-trigger delay must be added to the trigger delay when performance estimates are being made.

Digitizing time. When the instrument begins digitizing data, some time is required to complete the digitizing process. The time depends on the type of digitizer, the sample interval, the number of samples, and, in some cases, the sweep speed.

Digitizers fall into two basic classes—sequential and psuedo-random digitizers. Sequential digitizers acquire all data samples sequentially. The signal need net be repetitive because the complete waveform is acquired in a single pass.

Psuedo-random digitizers acquire their data in multiple passes or sweeps. If samples cannot be taken at a rate fast enough to acquire the waveform with sufficient resolution, the digitizer can acquire several cycles of a repetitive waveform. On each sweep, the samples are taken at a slightly different point on the waveform, so that after several sweeps enough samples are captured to accurately describe the input signal. Figure 6-5 illustrates the difference between sequential pseudo-random digitizing.

The acquisition time for a sequential digitizer is simple to calculate. It is just the number of samples times the sample interval. However, in a psuedo-random digitizer, samples are taken on several passes of a repetitive waveform. The acquisition is completed when a pre-defined number of points have been acquired. Thus, the acquisition time depends on the number of sweeps required to capture the points, the duty cycle of the waveform, and several other factors. The total digitizing time may be difficult to predict.

Signal averaging. Some digitizers offer the capability to repetitively acquire a waveform and average the data to remove random noise. This technique is very useful in many applications, but it is also time consuming. A complete acquisition is required for each average as well as time to compute the average. If a psuedo-random digitizer is used, the signal average may take quite a long time since every acquisition requires several sweeps and signal averaging requires multiple acquisitions.

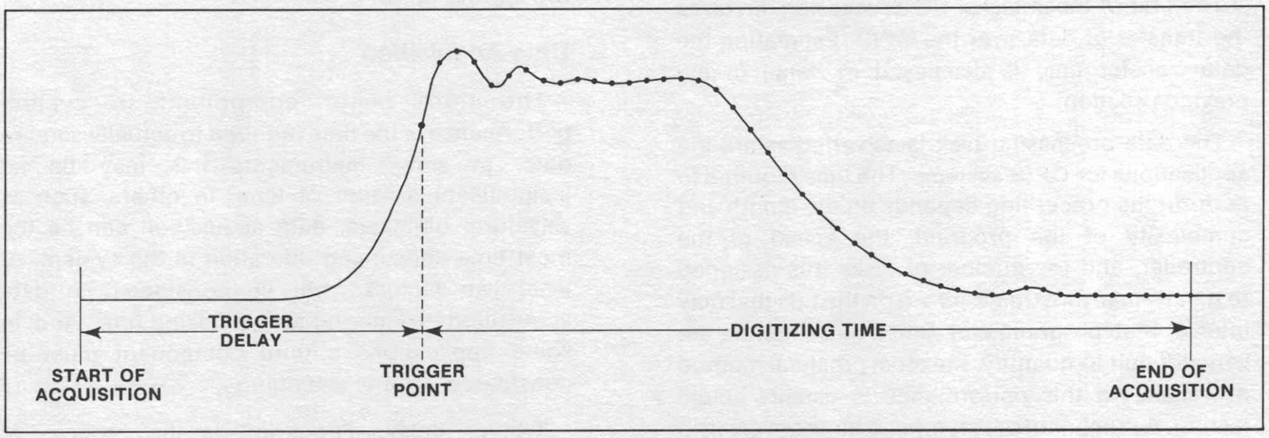


Fig. 6-4. *The total acquisition time is made up of trigger delay and digitizing time. In some digitizers, pre-trigger delay may also be added to this time.*

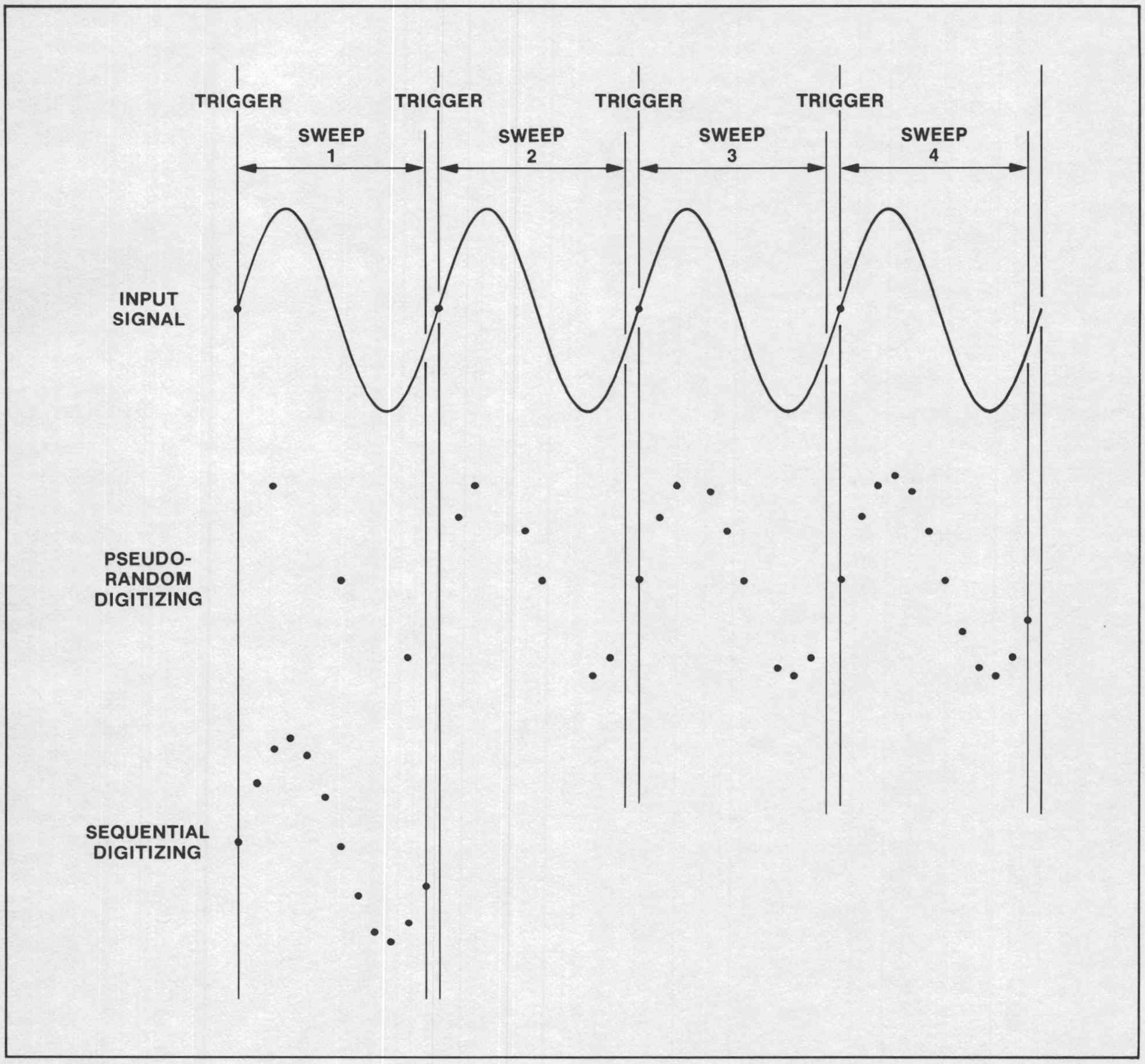


Fig. 6-5. *Psuedo-random digitizers sample a repetitive signal on several successive sweeps, gradually collecting enough samples to fully define the waveform. Sequential digitizers, on the other hand, acquire all the samples in a single pass. They must be capable of sampling the signal much faster than a psuedo-random digitizer.*

Human Interaction

Sometimes human intervention is required to enter parameters, make adjustments to non-programmable instruments settings, or set-up tests. Human interaction time is an important consideration if the system will require input or other assistance from an operator. This interaction can often be minimized with careful system design and the use of fully programmable instruments. If execution speed is a critical factor in the system design, the amount of human interaction should be minimized.

Section 7—Improving GPIB System Performance

Improving the performance of a GPIB system can be an elusive goal. The first step is to identify the components that affect the system performance and estimate how each component contributes to the overall performance. The previous section described each of these components and discused techniques for estimating the time required for each. This section provides some hints and techniques to improve system performance.

Know Your Instruments

To write efficient programs for controlling GPIB instruments, a good understanding of the instruments is essential. It's important to know how they buffer and execute commands and how the commands interact. Know the data formats used to transmit and receive data. And, if several formats are acceptable, know which ones the controller can transmit or receive fastest.

Also, know how the instrument acquires data. Does it respond to requests for waveform data while an acquisition is in progress? Does it signal the controller with an SRQ when data acquisition is complete?

Know the different conditions than can cause the instrument to assert SRQ. The programs you write will have to deal with any of these conditions that might occur during execution. Many instruments provide commands to disable specific interrupt sources, as well as the entire SRQ function.

The Operators or Programmers manuals should provide the specific information you need about the instruments. Read the programming and operation information carefully before beginning to write programs. A few minutes spent familiarizing yourself with the instruments can save hours of programming problems and frustrations, as well as produce a far more efficient system.

Choosing the Right I/O Statement

In many GPIB systems, data transfer takes a significant part of the total execution time. Significant performance improvements can be realized by carefully choosing the right I/O statements for each transfer and by carefully constructing the statement to minimize the amount of bus traffic.

PRINT. The PRINT statement provides a convenient means of transmitting ASCII data over the GPIB. It provides automatic addressing and unaddressing and data conversion and extremely flexible data formatting. Since the majority of GPIB instruments and peripherals communicate with ASCII, the PRINT statement is usually the best choice for sending device-dependent messages.

INPUT. The INPUT statement is the counterpart of PRINT for receiving data from the GPIB. It can receive data in a variety of formats and store it in string variables, numeric variables, or arrays. Like PRINT, the addressing and unaddressing functions are handled automatically. Most GPIB instruments transmit data and query responses in ASCII, so INPUT is usually the most efficient way to receive the data.

WRITE. The WRITE statement is designed to transmit data on the GPIB in 4052 internal binary format. Since few GPIB instruments understand this format, it is not particularly useful for instruments. However, it can be used to write data to GPIB storage peripherals. Since no data conversion is necessary on input or output, the transfer is considerably faster than other means. Addressing and unaddressing are handled automatically as with PRINT and INPUT.

READ. The READ statement is designed particularly to accept data in 4052 internal binary format. It is seldom useful for communicating with GPIB instruments, but can be used with GPIB storage peripherals. When data is written to a storage peripheral with WRITE, the READ statement can recover this data at high speed since no data conversion is necessary for internal binary format. Addressing and unaddressing is automatic.

WBYTE. The WBYTE statement is designed to provide low-level control of the GPIB for functions that cannot be implemented with the high-level statements. It gives you complete control over the GPIB data bus and the GPIB attention (ATN) and EOI lines. Any byte from -255 to +255 can be transmitted. However, WBYTE cannot transmit data from string variables—all data must be constants, numeric variables, or numeric expressions that reduce to a value of -255 to +255.

WBYTE is useful for transmitting binary data (not internal 4052 binary format), setting up peripheral-to-peripheral transfers, and sending interface messages. The added complexity of using WBYTE can be offset by considerable speed advantages of transferring data in binary rather than ASCII.

Absolute talk or listen addresses and secondary addresses must be specified with WBYTE. Unaddressing is also the programmer's responsibility.

RBYTE. The RBYTE (Read Byte) statement is the low-level counterpart of WBYTE. It accepts data from addressed talkers and assigns it to numeric variables. RBYTE can only receive numeric variables, and the largest value it can receive is 255 (the largest value that can be sent in a single byte). No addressing is required with RBYTE, because it assumes that a device has already been assigned to talk with WBYTE.

Minimizing Bus Traffic In PRINT

The PRINT statement provides an almost infinite variety of output formats. This infinite variety allows you to format the output data almost any way an instrument needs it. But, it also means that a slight difference in the format of the parameter list can make a big difference in the number of bytes added for formatting a message, and as a result, the transfer rate.

The most striking example is the use of the semicolon or comma delimiter between variables in the PRINT statement. If a semicolon is specified between variables or constants in the parameter list, each sample (variable or array element) is preceded by one space. Thus, one extra byte per sample is transmitted. If a comma is specified between variables, every sample is formatted into an 18-character field by adding spaces. For a typical four-digit data sample, 14 spaces are added, for a total of 18 bytes transmitted with each sample.

When array variables are transmitted, using the semicolon delimiter causes every element to be sent preceded by a single space. For example:

```
10 DIM A(100)
  .
  .
  .
50 PRINT @4:A
60 PRINT @4:A;
```

Transmitting four-digit data samples, the PRINT statement in line 50 transmits 5 bytes per sample compared to 18 bytes per sample in tranmitted by line 60. The result is that the statement in line 50 executes about 3.6 times faster!

If an instrument or peripheral requires a delimiter other than spaces between array elements, the PRINT USING statement can insert just about any delimiter or series of delimiters between the array elements.

A simple example is an instrument that requires each data sample delimited by a carriage return.

```
10 DIM A(100)
  .
  .
  .
50 PRINT @4:USING 100:A
  .
  .
  .
100 IMAGE 100(5D,/)
```

The PRINT statement in line 50 tells the 4052 to transmit array A using the format specified in the IMAGE statement of line 100. The IMAGE statement specifies that 100 five-digit numbers will be printed. All samples are formatted into five character fields by adding spaces. The slash indicates that each sample should be followed by a carriage return.

Any character can be used as a delimiter by replacing the slash in the above example with the desired character placed in quotes. For example, if a comma is required, the following statement could be used:

```
100 IMAGE 100(5D,",")
```

Be sure you know the format of the data when setting up the IMAGE statement. Attempting to transmit a value with more digits than specified in the IMAGE statement causes an error. Specifying too large a data field slows the transmission down because the samples are padded with spaces to fill the field.

The program in Fig. 7-1 generates an array of numbers from 1 to 100 and prints each value, delimited by commas, on the 4052 screen. Using a carefully constructed IMAGE statement, extra "padding" bytes are eliminated, increasing the data transfer rate. Though the values are only printed on the screen in this example, the same technique applies to tranmitting data over the GPIB.

```
10 DIM A(100)
20 FOR I=1 TO 100
30 A(I)=I
40 NEXT I
50 PRINT USING 60:A
60 IMAGE 99(FD,","),FD

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,
43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,
62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,
81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,
100
```

Fig. 7-1. *A simple program illustrating the use of the PRINT USING statement. The output printed on the 4052 screen is also shown.*

The first four lines of the program generate the array. Then, line 50 prints the array on the graphic system display with each element delimited by a comma. The "FD" field operator in the IMAGE statement (line 60) prints the values in fields just large enough for the number with no added spaces. For the first nine samples (1-9), a single digit field is printed. The next 90 samples (10-99) are printed in two-digit fields, and the last sample (100) is printed in a three-digit field. All samples except the last one are followed by a comma.

Synchronizing the Controller and Instruments

When a GPIB instrument is operating under program control, two programs are running, not just one. One program is running in the controller, and another in the microprocessor system in the instrument. It is important that these two programs be sychronized. Fortunately, synchronizing the programs is easy with Tektronix instruments.

Program execution in a Tektronix programmable instrument is controlled by the messages it receives from the GPIB. The details of how the message is processed vary slightly between instruments, but from the user's point of view they are all the same. Messages go into an input buffer as they are received. If the message is too long to fit in the buffer, the instrument automatically holds off further input and begins processing the message. Several other conditions, such as receiving the message terminator can also start the message processing. In any case, while the instrument is busy processing the message, it sets bit 5 (the busy bit) in its status byte and asserts $\overline{\text{NRFD}}$ (Not Ready For Data) to hold off further input.

While $\overline{\text{NRFD}}$ is asserted, the 4052 cannot send any more messages to the instrument or get any output from it. To illustrate, the following program repeatedly sends the FREQ command to a 492P Spectrum Analyzer incrementing the value on each pass through the loop.

```
100 FOR I=1 to 10
110 PRINT @1:"FREQ ";I;"GHZ"
120 NEXT I
```

This loop is executed much slower than it would if the message in line 110 were simply printed on the display. That's because the 492P will not accept the message until the message from the last loop is processed. Thus, the 4052 waits at line 110 on each loop for the 492P to finish execution of the last message.

This also works for input from an instrument. The program below increments the center frequency for a 492P just like the last one did, but it also asks for the frequency range at the end of each message.

```
100 FOR I=1 TO 10
110 PRINT @1:"FREQ ";I;"GHZ;FRQRNG?"
120 INPUT F(I)
130 NEXT I
```

On each pass through the loop, the FREQ command is sent and the frequency range is queried. Immediately after the message is sent in line 110, program control advanced to line 120. The talk address for the 492P is sent, but it does not begin talking until the message in line 110 is completed and the frequency range query response is ready.

Interrupts Boost Performance

Using the operation complete interrupt. A large part of a system's execution time can be consumed by waiting for asynchronous events, such as the completion of an acquisition. If the system controller is tied up waiting for these events, the performance of the system can be seriously degraded. The SRQ interrupt feature of many GPIB instruments can be used to minimize this waiting time and therefore, improve the performance of the system.

Consider, for example, a system consisting of a 4052 and one or more 7612D Programmable Digitizers. A simple program to acquire a waveform from the 7612D is shown in Fig. 7-2.

```
10 DIM D(512)
20 PRINT @1,0:"ARM A;READ A"
30 WBYTE @65,96:
40 RBYTE H,B0,B1,D,C0
50 WBYTE @95:
```

Fig. 7-2. *A simple program to acquire a waveform from a 7612D Programmable Digitizer.*

This program sends the 7612D an ARM command to prepare channel A to acquire data. When a trigger is received, the acquistion begins. Until the acquisition is complete, channel A data cannot be read.

Immediately following the ARM command is a READ command. Since the channel A data cannot be read, the 7612D buffers the command and waits for the acquisition to complete. Line 30 assigns the 7612D to talk, but since data cannot be read from channel A yet, the 7612D holds the bus in a waiting state until the acquisition is complete. As a result, program execution is held at line 40, waiting for the 7612D to begin transmitting its data.

During this waiting period, no device-dependent message traffic can occur and no other processing takes place—the time is essentially wasted. If the 4052 could be freed to perform other tasks during this period, the system performance could be significantly improved. The problem is how to signal the 4052 when the acquisition is complete so that the process of reading the data can be initiated.

That's where SRQ interrupts come in. The 7612D, like most other acquisition instruments from Tektronix, have an operation complete or waveform readable interrupt feature that allows them to generate an SRQ when data becomes available for the controller. The program shown in Fig. 7-3 is taken from the previous one with modifications added to use the waveform readable interrupt feature of the 7612D.

This program gets under way by setting up an ON statement for the SRQ interrupt. Then, line 20 sends the WRI ON and ARM A commands to the 7612D. These commands enable the WRI (Waveform Readable Interrupt) and arm channel A for acquisition. At this point, the 4052 is free to perform other tasks, such as process data from the last acquisition or service other instruments.

When the 7612D completes the acquisition, it asserts SRQ. This interrupts the currently executing

```
10 ON SRQ THEN 200
20 PRINT @1,0:"WRI ON;ARM A"
30 REM *** PROGRAM LINES TO PERFORM OTHER ***
40 REM *** SYSTEM TASKS ARE ADDED HERE    ***
   .
   .
   .
   .
200 POLL X,Y;1
210 IF Y<>197 AND Y<>199 THEN 260
220 PRINT @1,0:"READ A"
230 WBYTE @65,96:
240 RBYTE H,B0,B1,D,C0
250 WBYTE @95:
260 RETURN
```

Fig. 7-3. *Adding a few lines to the program in Fig. 7-2 takes advantage of the waveform readable interrupt feature of the 7612D.*

routine and causes control to be passed to the routine starting at line 200. Line 200 polls the 7612D (other instruments could be added to the POLL list, but are omitted for clarity). If the status byte is either 197 or 199 (decimal), a waveform readable interrupt for channel A has occurred and the binary data is read in lines 220 through 250. Otherwise, control is passed directly to the RETURN statement and the interrupt is ignored.

The performance improvement gained with this technique is a function of the instrument's acquisition time. For example, if the 7612D is acquiring a 2048-point waveform at one millisecond sampling interval, the total acquisition time is over 2 seconds (excluding pre-trigger delay time). For faster sampling intervals or waveforms with fewer points, the gain will be less dramatic.

Prioritizing serial poll response. When a system consists of several instruments that can assert SRQ, it is possible that more than one instrument will assert SRQ at a time. It may be more important to respond to an SRQ from one instrument or group of instruments than from others in the system. For example, a digital voltmeter might take a new sample every two seconds and assert SRQ each time it does. Since the sample will be overwritten by a new one every two seconds, it's important to respond to the DVM's interrupt and read the data before the next sample is taken.

If the system also contains other devices whose interrupts don't need such timely response, the response to these interrupt must somehow be prioritized so that the most important devices are serviced first. Prioritizing the serial poll response is simple—just list the devices numbers in the POLL statement in priority order from highest to lowest priority. If the DVM in the previous example is the highest priority device in the system, it should be listed as the first device in the POLL address list, followed by the next most important device, and so on until all devices that can assert SRQ are listed.

For example, Fig. 7-4 shows an interrupt handling routine for a system with a digital voltmeter (DVM), a function generator, and a magnetic tape drive.

```
100 POLL D,S;4;1;3
110 GOSUB D OF 130,200,300
120 RETURN
130 REM *** SRQ HANDLING FOR THE DVM ***
  .
  .
  .
190 RETURN
200 REM *** SRQ HANDLING FOR THE FUNCTION GEN. ***
  .
  .
  .
290 RETURN
300 REM *** SRQ HANDLING ROUTINE FOR THE MAG TAPE DRIVE ***
  .
  .
  .
390 RETURN
```

Fig. 7-4. *An interrupt handling routine for a GPIB system with a DVM, function generator, and tape drive.*

When an SRQ occurs, program control is transferred to line 100. The POLL statement begins a serial poll process, starting with the first device in the list. Since the DVM's interrupt is the most important, it is checked first. If it is asserting SRQ, it is serviced first even if other devices are also asserting SRQ. If the DVM is not asserting SRQ (indicated by setting bit 7 of the status byte), the next device in the list is polled, and so on until the device that is asserting SRQ is found.

Local Data Processing

Many GPIB instruments are capable of performing some on-board processing. This processing may be faster than sending data over the bus to the controller and performing the processing in the controller, especially if the processed data must be shipped back to the instrument for display or further processing.

A good example is the signal average function provided on many instruments. To signal average in the controller, each waveform must be sent to the 4052. If a large number of waveforms are averaged, the time to transmit the waveforms can become quite substantial. If the same function is performed in the instrument, the data can be acquired and averaged and a single averaged waveform sent to the controller. In most cases, this process is considerably faster than performing the average in the 4052.

Another example is illustrated in the two programs shown Fig. 7-5. Part **a** of the figure shows a program that acquires a spectrum from a 492P Spectrum Analyzer and finds its maximum value using the 4052R07 ROM Pack MAX function. The program in part **b** of the figure accomplishes the same function using the 492P FMAX and POINT commands.

```
10 PRINT @1:"CURVE?"
20 INPUT A
30 CALL "MAX",A,M,P
```

a. *Finding the maximum value of the waveform in the 4052.*

```
10 PRINT @1:"FMAX;POINT?"
20 INPUT M,P
```

b. *Finding the maximum value of the waveform in the 492P.*

Fig. 7-5. Two programs to find the maximum value of a digitized spectrum acquired by the 492P. The program in part **a** *performs the operation in the 4052, while the program in part* **b** *performs the same operation in the 492P.*

The program in part **a** executes considerably slower because it takes about nine seconds to transmit 1000 points of ASCII data to the 4052. (This time could be reduced to about 350 milliseconds if the data is transmitted in binary.) In the right program only a single point is transmitted since the 492P finds the maximum value itself and transmits it to the 4052. As a result, this technique is much faster.

ASCII vs. Binary—Simplicity vs. Speed

GPIB instruments and controllers usually transfer numeric data such as waveforms in one of two formats—ASCII-coded decimal numbers or binary numbers. Sometimes you have no choice which format to use because the instruments or controller require one format. Other instruments, like the 492P Programmable Spectrum Analyzer can transmit data in either format.

The question in this case is, which format should I use? There are good reasons to choose each. ASCII data transmission is usually simpler. But, binary data transmission is significantly faster.

In the 4052, ASCII data can be transmitted and received using simple PRINT and INPUT statements. Data storage format is also more flexible with ASCII. Numeric data may be stored in string variables or numeric variables. A wide variety of input terminators are also available to simplify breaking the data into manageable blocks.

However, ASCII data transmission is slow compared to binary transmission. In ASCII, a single numeric value is converted to a string of ASCII characters before transmission (Fig. 7-6). For example, the value 237 is converted to the ASCII code for "2", followed by the ASCII code for "3" and the ASCII code for "7". Usually, a delimiter character (such as space or comma) is appended before or after the string, for a total of four bytes.

Binary data transmission is considerably faster, since a one or two-byte binary equivalent of the number is transmitted. Any value between 0-255 can be transmitted in a single byte, and any value between 0-65,535 can be transmitted in two bytes. However, binary transfers are more complex to implement. Data is always stored in numeric variables and if the value is larger than 255, it requires two variables—one for each byte.

Though the low-level RBYTE and WBYTE statements in 4050 BASIC transfer data slower than the PRINT and INPUT statements, the number of bytes sent is drastically reduced, which usually more than compensates for the slower transfer rate.

To get an idea of the speed difference between ASCII and binary data transfers, refer to Table 7-1.

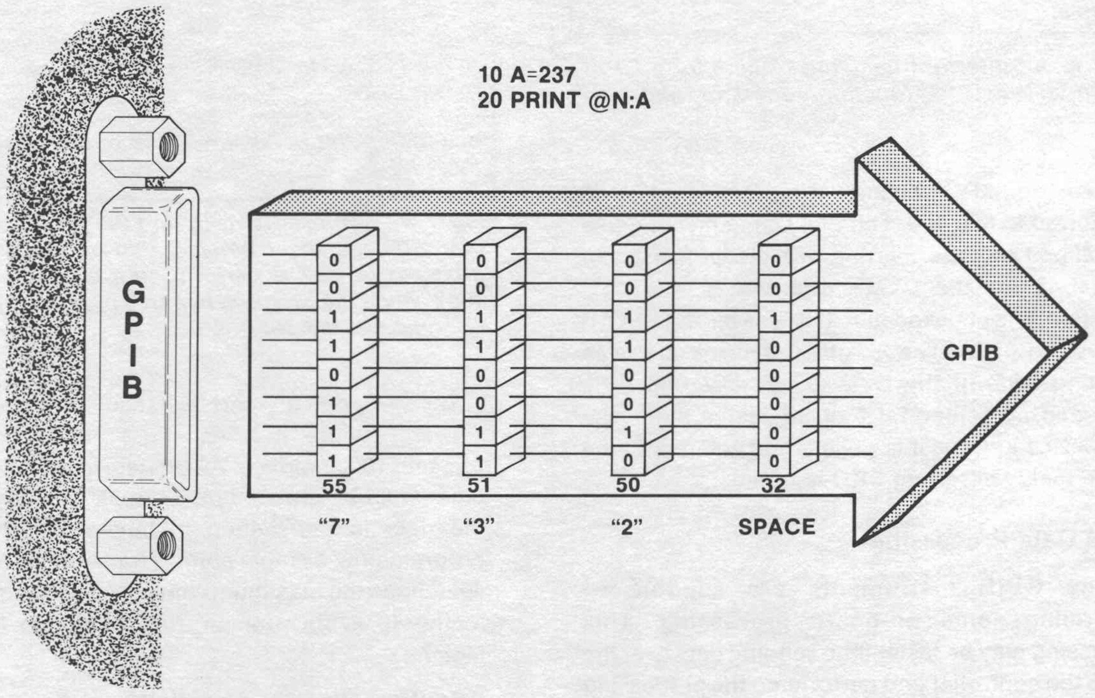


Fig. 7-6. *Transmitting a numeric value in ASCII. The addressing sequence is omitted for clarity.*

This table shows some typical times to transfer a 1000-point waveform from the 492P to the 4052 in both ASCII and binary format. It's easy to see from this example why a slightly more complex program is a small price to pay when speed is important.

**TABLE 7-1
WAVEFORM TRANSFER TIMES FOR
492P/4052 SYSTEM**

Binary transfer	350 ms
ASCII (input to string)	8.5 sec
ASCII (input to numeric array)	9.0 sec

In some cases, binary data may require some special processing to put it into a useable format. The time required to perform this processing must also be considered when choosing data transfer mode.

For example, some instruments send binary data in two bytes for each point. The controller has to assemble these two bytes into a single numeric value before the data can be processed. If performing this process requires a significant amount of time, it may be more efficient to transfer data in ASCII, even though the data transfer itself is slower.

Data Logging

In some cases, there simply isn't enough time to perform all the necessary processing on acquired data at the time it is acquired. A common solution to this problem is data logging, where data is captured and stored on a peripheral device at high speed. Then, after the acquisition is complete, the data is processed at a slower rate.

The WBYTE statement allows the 4052 to set up transfers between an acquistion instrument, for example, and a peripheral storage device. The process of setting up such a transfer is described in **Transfers among GPIB devices** in Section 4.

Appendix A—Subsets Describe Interface Functions

The IEEE 488 standard allows a designer a great deal of flexibility in implementing an instrument's GPIB interface. The functional capability of the interface is divided into ten interface functions. From this list of functions, a designer may choose to implement all, part, or none of each function, as defined by the interface subsets listed in the standard.

The standard also specifies a shorthand way of describing which optional functions are implemented in a device's interface. Each function is assigned a mnemonic (e.g. AH for Acceptor Handshake, DT for Device Trigger). A number appended to the end of the mnemonic indicates how many, if any, of the optional features of that function are implemented. Zero indicates that the function is not implemented at all. SH0, for instance, means that no source handshake capability is implemented in the interface. SH1 means that full source handshake capability is implemented.

Several of the interface functions have only two options—full capability or no capability. Others, such as the talker and controller functions, offer many options. Table 1 shows a summary of the interface subsets with a brief explanation of each function. The controller function is not included in the chart since it contains an extensive list of options. Refer to the IEEE 488 standard for information on controller interface subsets.

It's important to remember that the interface subsets describe the repertoire of the interface only. They don't say anything about the programmable functions of the instrument. But, they're still important, because the programmable features of the instrument can't be used to full advantage without the appropriate interface functions. For example, if an instrument sends data over the bus, the talker and source handshake functions must be implemented in its interface. However, the designer may choose from one of nine levels of basic talker capability or nine levels of extended talker capability. The choice is based on which of the optional functions are required. If the instrument implements the T7 talker subset, it will have basic talker capability with talk-only mode, no serial poll capability and it will not remain a talker when addressed to listen.

The key is knowing which interface subsets your application requires. The interface subsets are usually listed in the specifications for GPIB instruments. Some instruments even print the subsets on the panel near the GPIB connector.

Appendix A
Subsets Describe Interface Function

SOURCE HANDSHAKE		SH0	SH1
Full capability	Allows a device to generate the handshake cycle for transmitting data		X
No capability		X	

ACCEPTOR HANDSHAKE		AH0	AH1
Full capability	Allows a device to generate the handshake for receiving data		X
No capability		X	

TALKER (EXTENDED TALKER)*		T0 (TE0)	T1 (TE1)	T2 (TE2)	T3 (TE3)	T4 (TE4)	T5 (TE5)	T6 (TE6)	T7 (TE7)	T8 (TE8)
Basic Talker (Basic Extended Talker)	Allows an instrument to transmit data		X	X	X	X	X	X	X	X
Talk Only Mode	Allows an instrument to transmit data without a controller on the bus		X		X		X		X	
Unaddressed If My Listen Address (MLA)	Prevents an instrument from being a talker and a listener at the same time						X	X	X	X
Serial Poll	Allows an instrument to send a status byte in response to a serial poll		X	X			X	X		
No capability		X								

LISTENER (EXTENDED LISTENER)*		L0 (LE0)	L1 (LE1)	L2 (LE2)	L3 (LE3)	L4 (LE4)
Basic Listener (Basic Extended Listener)	Allows an instrument to receive data		X	X	X	X
Listen Only Mode	Allows an instrument to receive data with a controller on the bus		X		X	
Unaddress if My Talk Address (MTA)	Prevents an instrument from being a talker and a listener at the same time				X	X
No capability		X				

SERVICE REQUEST		SR0	SR1	
Full capability	Allows an instrument to request service from the controller with the SRQ line		X	
No capability		X		

REMOTE-LOCAL		RL0	RL1	RL2	
Basic Remote-Local	Allows the instrument to switch between manual (local) control and programmable (remote) operation		X	X	
Local Lock-Out	Allows the return to local function to be disabled		X		
No capability		X			

PARALLEL POLL		PP0	PP1	PP2	
Basic Parallel Poll	Allows an instrument to report a single status bit to the controller on one of the data lines (DI01-DI08)		X	X	
Remote configuration	Allows the instrument to be configured for parallel poll by the controller		X		
No capability		X			

DEVICE CLEAR		DC0	DC1	DC2	
Basic Device Clear	Allows all instruments on the bus to be initialized to a predefined state cleared		X	X	
Selective Device Clear	Allows individual instruments to be cleared selectively		X		
No capability		X			

DEVICE TRIGGER		DT0	DT1	
Full capability	Allows an instrument or group of instruments to be triggered or some action started upon receipt of the Group Execute Trigger (GET) message		X	
No capability		X		

** Extended talkers and listeners use secondary addresses; other talkers and listeners do not.*

NOTES